# Introduction to OpenACC

Alexander Fu, David Lin, Russell Miller

June 2016

Taking advantage of the processing power of the GPU is what makes CUDA relevant. However, using CUDA and constraining oneself to solely NVIDIA GPUs can be limiting. The directives in OpenMP lack the ability to utilize GPUs, but make parallel programming both elegant and concise. OpenACC accommodates for these weaknesses. This guide should serve as an introduction on how to use OpenACC.

## Contents

# 1 Overview

OpenACC is by strict definition, "a sect of standardized, high-level pragmas that enables C/C++ and Fortran programmers to utilize parallel coprocessors." OpenACC was developed initially by PGI, Cray, CAPS enterprise, and NVIDIA with the purpose of providing a standard for accelerator directives. True to its name, OpenACC serves as an **Accelerator Programming** API. The term **Accelerator Programming** describes the general process of offloading functions from the CPU to the specialized hardware such as GPUs, and coprocessors.

One benefit of OpenACC would be that it takes a higher level approach, similar to that of OpenMP, where acceleration is achieved through a series of programmer directives, or pragmas. This extra level of abstraction brings many benefits, ranging from code-readability to generally faster code due to compiler optimizations.

An advantage of OpenACC over CUDA would be the support for other coprocessors other than NVIDIA GPUs. A wide variety of additional coprocessors, such as AMD GPUs and Intel MICs, are supported, consequently making OpenACC portable to most hosts / coprocessor combinations. In the case that a compiler does not support OpenACC, the pragma directive will simply be ignored and the program will continue to run, albeit serially.

As an aside, from the previous points it appears that OpenACC essentially provides a more "general" version of OpenMP, which brings the practicality of OpenMP to light. To address this, there are efforts and plans in place to merge the two approaches together.

## 2 Combining Approaches

In the case of OpenMP and MPI, both approaches can be combined with OpenACC. In the case of OpenMP however, the directives cannot mix, i.e. directives cannot be used in the same loop at the same time.

Combining CUDA and OpenACC are fully interoperable. This has many benefits, as the programmer can develop using the OpenACC higher level directives, and implement the very computational complex functions manually using CUDA.

## 3 The Setup

There are several ways to compile OpenACC programs. It can be compiled using gcc (given the proper configuration), as well as several commercial compilers such as, PGI and Omni. For our purposes, we chose to use Omni, which worked with little to no hassle. The only downside was that the OpenMP definitions were a bit outdated, so the OpenMP pragmas you might be familiar with could possibly throw compile errors.

When compiling an OpenACC program, there are several dependencies that must be installed and located. We will need **MPI**, **NVCC** for compilation, and the location of the shared library lcudart.so for running your program.
For your convenience, the following are the commands that we used to set up the compilation. Your mileage may vary.

```
1 setenv OMPCC /path_to_ompcc_compiler/
2 setenv MPI /path_to_mpich_folder/
3 setenv CUDA /path_to_cuda-5.5_folder/
4 setenv PATH ${PATH}:${MPI}/bin:${OMPCC}/bin:${CUDA}/bin
5 setenv LD_LIBRARY_PATH ${CUDA}/targets/x86_64-linux/lib
```

OpenACC has a few command line options. The most useful are detailed below. The rest can be found by running ompcc with the -h flag.

| Useful Command Line Options | | |
|---|---|---|
| Option | Description | Usage |
| -o | Same as gcc's -o. Sets the output filename. | -o out_file.extension |
| -c | Same as gcc's -c. Compile only. Don't link. | -c |
| -h | Displays the various command line options. | -h |
| --debug | Saves all intermediate files to a s-ubfolder. | --debug |

It is often interesting to see the results of automatic parallelization done by the OpenACC compiler. You may find the following makefile useful.

```
CC      = /path_to_ompcc_executable
CCFLAGS = -O3 # Optimize the program even more.
ACCFLAGS = -acc --debug # You can see the generated code!.
BIN = program_nve program_acc

all: $(BIN) # If Make is invoked without a target, make both versions.
program_acc: program.c
   $(CC) $(CCFLAGS) $(ACCFLAGS) -o $@ $<
program_nve: program.c
   $(CC) $(CCFLAGS) -o $@ $<
clean:
   $(RM) $(BIN)
```

# 4 Measuring Success

In a lot of cases, you may not know if the naive or OpenACC will be better. Therefore, we must set up some way to measure. As a general rule, bigger data sets are better measures of efficiency, due to the overhead of parallel setup. Here is the timing code we use to measure the number of milliseconds elapsed.

```
#include<time.h>
...
//In the body of a function
clock_t t1, t2;
t1 = clock();
your_function_call();
t2 = clock();

printf("Elapsed Time: %f ms\n",((float)(t2 - t1) / (float)CLOCKS_PER_SEC ) * 1000);
```

Let's get started!

# 5 The `kernels` Construct

OpenACC allows us to use this directive to give the compiler more information about a particular piece of code. This keyword is usually used to describe a structured block of code to be run on an accelerator (aka the GPU in this case). The compiler will try to optimize the code in this section if possible. By wrapping blocks of code in this pragma, you are essentially writing the .cu CUDA code that is to be run on the kernel. This is an abstraction, making memory allocation and management easier.

# 6 My First Program

We start with some simple matrix operation. We take a large square matrix (10000 x 10000), fill it with random numbers between 0 and 99, and have the parallel program double each element in the matrix.

**IMPORTANT TO NOTE:** We want to exploit cache locality, and so we have tried to make the matrix a single contiguous block of memory. Remember that the equivalent access to the $i$th row, $j$th column element is an access to the $(i * width + j)$-th element in our array, which holds for row-major programming languages like C. Though we could access the matrix with only one for loop, we have left the conventional double for loop for easier adaptation to operations like multiplication.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MATRIX_SIZE 10000

int main() {
    int randomMatrix[MATRIX_SIZE * MATRIX_SIZE];
    //Set the seed
    srand(time(NULL));
    for(int i = 0; i < MATRIX_SIZE; i++)
        for(int j = 0; j < MATRIX_SIZE; j++)
            randomMatrix[i * MATRIX_SIZE + j] = rand() % 100;

    clock_t t1, t2;
    t1 = clock();
    int i, j;
    #pragma acc kernels
    {
    for(i = 0; i < MATRIX_SIZE; i++)
        for(j = 0; j < MATRIX_SIZE; j++)
            randomMatrix[i * MATRIX_SIZE + j] = randomMatrix[i * MATRIX_SIZE + j] * 2;
    }
    t2 = clock();
    printf("Elapsed Time: %f ms\n",((float)(t2 - t1) / 1000000.0F ) * 1000);
    return 0;
}
```

Just like OpenMP, we have special **pragmas**. The kernels pragma gives the compiler free rein in finding parallelization. For our purposes, this seems perfect! We just want to see what kind of speedup OpenACC provides.

Let's run it! We find that... Oh. What a twist. The non-parallel version takes half a millisecond, while the generated CUDA version takes more than 282 milliseconds! How can this be? In our experiences with OpenMP and CUDA, we usually found CUDA to be faster for independent parallel matrix element manipulations, since each thread would be responsible for exactly one element!
To find the answer, we must dive into the generated code. Remember above when we mentioned the --debug flag? Insert it into your Makefile rule for generating the CUDA version of the program, Make it, and look inside the newly generated __omni_temp__ folder. You should see something like program_name_pp.cu, which you can open to take a look. The important portion is edited and listed here.

```
__global__ static
void _ACC_GPU_FUNC_main_L28_DEVICE(int *j, int *i, int randomMatrix[100000000])
{
    int _ACC_loop_iter_i;
    for((_ACC_loop_iter_i)=(0);(_ACC_loop_iter_i)<(10000);(_ACC_loop_iter_i)+=(1))
    {
        int _ACC_loop_iter_j;
        for((_ACC_loop_iter_j)=(0);(_ACC_loop_iter_j)<(10000);(_ACC_loop_iter_j)+=(1))
        {
            if((_ACC_block_x_id)==(0)){
            (randomMatrix[((_ACC_loop_iter_i)*(10000))+(_ACC_loop_iter_j)])
                =((randomMatrix[((_ACC_loop_iter_i)*(10000))+(_ACC_loop_iter_j)])*(2));
            }
        }
        if((_ACC_thread_x_id)==(0)){
            (*j)=(_ACC_loop_iter_j);
        }
        _ACC_GPU_M_BARRIER_THREADS();
    }
    if((_ACC_thread_x_id)==(0)){
        (*i)=(_ACC_loop_iter_i);
    }
    _ACC_GPU_M_BARRIER_THREADS();
}
```

Uh oh. We might have found our problem. Rather than each thread changing one element, it looks like only one thread is handling **all** of the matrix. This is a classic case where we are over-estimating OpenACC's compiler. Just because we know of a great way to do a problem, that doesn't mean that the automatic parallelization capabilities of OpenACC can know what we are thinking and accommodate. OpenACC is a program that is generic enough to spot parallelization opportunities, but that does not mean these opportunities are always optimal. In this example, each element was accessed exactly once. If we take into account the host-to-device and device-to-host memory transfer operations as well, it makes sense as to why this version is so slow.

So what can we do to make the program faster?

# 7 The `loop` Construct

Much like OpenMP's `for` construct, this instruction tells the compiler to break up the loop into many smaller chunks, so that parallelism can be achieved. Nested for loops can also have this pragma applied to multiple loops. An important thing to note is that when used with the `kernels` construct, this makes the entire section into kernel code. You should see similarities between this OpenACC code and similar OpenMP code. For example the two snippets below are the same:

```
1 #pragma acc kernels
2 {
3     #pragma acc loop
4     for(...)
5     {
6         ...
7     }
8 }
```

```
1 #pragma kernels acc loop
2 for(...)
3 {
4     ...
5 }
```

Because of this, it should make sense as to why only the outer loop below has the `kernels` keyword:

```
1 #pragma kernels acc loop
2 for(...)
3 {
4     #pragma acc loop
5     for(...)
6     {
7         ...
8     }
9 }
```

## 7.1 The `independent` Construct

This is a pretty simple one. By using this construct in conjunction with the `loop` construct, we can tell the compiler that loop iterations are data-independent and can be executed in parallel, overriding compiler dependency analysis. We can apply this to our program, and use the `loop` construct for both loops. We replace lines 19 through 24 in section 6 with the following:

```
19 #pragma acc kernels loop independent
20 for(i = 0; i < MATRIX_SIZE; i++) {
21     #pragma acc loop independent
22     for(j = 0; j < MATRIX_SIZE; j++) {
23         randomMatrix[i * k + j] = randomMatrix[i * k + j] * 2;
24     }
25 }
```

If we look at the generated CUDA code, it looks a little closer to what parallelism should look like. This is the outer loop:

```
... //Various block calculations
for((_ACC_block_x_idx)=(_ACC_block_x_init); (_ACC_block_x_idx)<(_ACC_block_x_cond);
    (_ACC_block_x_idx)+=(_ACC_block_x_step))
{
    ... //Various calculations
    {
        ... // Other nested for loop
    }
}
```

Here we can clearly see that each thread's code will start at a generated _ACC_block_x_init, go a certain distance, and increment to the next chunk using the _ACC_block_x_step. If we run this code, we get a runtime of 10 ms. It's still 20x slower than the naive, but it's 28x faster than our original CUDA program! Perhaps the generic `kernels` construct is a little too generic. How do we tackle this?

# 8 The `parallel`, `vector`, `gang`, and `worker` Constructs

## 8.1 parallel

Rather than leaving all of the optimization (which could be less efficient) to the compiler, the `parallel` construct (much like from OpenMP's construct of the same name) offers more fine-grained control on pieces of code. Generally speaking, `kernels` instructs the compiler to optimize a piece of code to produce an arbitrary number of kernels using arbitrary dimensions of grids and blocks, in many cases changing these dimensions between different kernels. Meanwhile, the `parallel` construct usually uses a constant number of grids and blocks (and warps). However, this `parallel` construct can be more explicit, but requires more analysis by the programmer to determine the exact dimensions.

```
#pragma acc parallel
{
    ... //just like OpenMP's parallel. Makes an instance per thread.
}
```

## 8.2 vector, gang, worker

The `vector, gang,` and `worker` constructs correspond to CUDA terms. Specifically, a vector is like a thread, a gang can be thought of as a thread block, with a worker as a warp. For our purposes it is easier to not deal with workers. You can specify the dimensions per parallel code block, like this:

```
#pragma acc parallel vector_length(128) num_gangs(1)
{
   for (...)
}
```

Which will generate code that looks like this:

```
extern "C" void _ACC_GPU_FUNC_0( ... )
{
   dim3 _ACC_block(1, 1, 1), _ACC_thread(128, 1, 1);
    ...
}
```

As you can see, the number of threads equals the vector length, and the number of gangs equals the number of blocks. Each thread can calculate the value of its loop variable from a virtual index. the constructs to set these values are `vector_length`, `num_gangs`, and `num_workers`. You can then use the keywords `vector`, `gang`, and `worker` to specify how to share iterations of the loop across regions.

```
#pragma acc parallel loop vector_length(128) num_gangs(1) gang
for (...)
{
    #pragma acc loop vector
    for(...)
}
```

For example, the outer for loop not only defines the dimensions but also that this loop should be shared by gangs. The inner loop instructs the kernel to split it across the vector (threads).

# 9 The `reduction` Clause

This is a simple one. If we want to "reduce" a variable like in OpenMP, the syntax is the same, and must be used with the `parallel` construct.

```
int x = 0;
#pragma acc parallel loop reduction(+:x)
for (...)
```

Using the above code, each thread would have its own copy of x. When the block finishes executing, all threads' x's will be summed into the main thread's x.

# 10 The Memory Constructs

You can also specify the memory copying from device to host and vice-versa in an abstracted fashion. You can also choose to move memory independent of parallel sections by using the `data` construct. What's interesting is that you can also specify which elements to copy as well.

## 10.1 copy

```
1  int matrix[10];
2  int n = 5;
3  ...//fill matrix somehow
4
5  #pragma acc data copy( matrix[0:n] )
6  {
7      ...
8  }
9
10 #pragma acc kernel copy( matrix[0:n] )
11 {
12     ...
13 }
14
15 #pragma acc parallel copy( matrix[0:n] )
16 {
17     ...
18 }
```

The above code is quite simple. Upon executing line 5, space for (n-0) ints is created on the device. The matrix, despite being 10 elements long, only has the first five elements (because of the slicing) copied into the device from the host. On line 8 at the "}", the end of the pragma, the device copies the memory back into the host. The space is then freed from the device. Lines 10 and 15 also show that this construct can be used with kernel and parallel sections respectively.

## 10.2 create

If you were to replace all instances of `copy` with `create` on the previous page, the program would allocate space on the device only. No memory gets copied in or out. This is useful for when you need a temporary scratch space for calculations.

## 10.3 copyin

If you were to replace all instances of `copy` with `copyin` on the previous page, the program would allocate space on the device, and like the name suggests, copy in the memory from host to device. In some cases (like matrix multiplication), you might want the output to be a different variable because of differing dimensions from the input. The memory on the device is not copied back to the host after the pragma block. In this case, you would only need to copyin the data but you would never need to retrieve the data back again.

## 10.4 copyout

Allocates space on the device (but does not transfer memory from host to device!), and transfers the data in the space to the host after the end of the pragma block. This is useful for when you know the dimensions of the output, and can write the final values of the memory immediately.

### 10.5 Combining

We can actually use most of these instructions for a simple example. This section should also serve as usage instructions for using multiple memory pragmas. Suppose we want to multiply matrices A, B, C, and output into a matrix D. We will need scratch space, which we call AB, to hold the result of the multiplication of A and B. We do not need to copy in or out this AB, because it was only the intermediate step. A B and C's values don't change after the operation, so we can read them into the GPU memory without transferring them back from device to host. In the case of D, we want to allocate memory, but we don't want to copy anything from host to device, since we will be filling in the matrix with the correct values ourselves, but we do want to extract it from the device. Therefore, the data setup should read:

```
1 int A[10], B[10], C[10], D[10];
2 int n = 10;
3 ...//fill matrixes somehow
4 #pragma acc data copyin(A[0:n]) copyin(B[0:n]) copyin(C[0:n]) create(AB[0:n])
      copyout(D[0:n])
5 {
6     ... // Matrix mult code
7 }
```

## 11 My First Program Version 2

Here is the code using everything we've learned so far. We know that we will need one thread for each element, and with 1000 * 1000 elements. Remembering the CUDA limits, we should probably split the work into 62500 gangs and 16 vectors!

```
19 #pragma acc parallel loop independent copy(randomMatrix[0:MATRIX_SIZE * MATRIX_SIZE])
       vector_length(16) num_gangs(62500) gang
20 for(i = 0; i < MATRIX_SIZE; i++) {
21     #pragma acc loop independent vector
22     for(j = 0; j < MATRIX_SIZE; j++) {
23         randomMatrix[i * MATRIX_SIZE + j] = randomMatrix[i * MATRIX_SIZE + j] * 2;
24     }
25 }
```

It still takes 10 ms, but we suspect that is due to the memory copying. Now that we know most of the basics, you can find two commented full programs in appendix A and B, as well as a segmented non-negative matrix factorization in appendix C.

## 12 Sources and Additional Readings


https://developer.nvidia.com/openacc
https://gcc.gnu.org/wiki/OpenACC
http://omni-compiler.org/openacc.html
https://www.pgroup.com/resources/accel.htm
http://www.drdobbs.com/parallel/easy-gpu-parallelism-with-openacc/240001776

# A Sample Program 1

Suppose you are given a square matrix of floats ranging from 0 to 1. If a square of k x k elements all are over a threshold t, then this is considered a "bright spot." Your job is to determine how many bright spots are in a certain matrix. Let's be evil and say that there will be the maximum number of elements over threshold, but our program should not know this. Therefore we pick the threshold to be 0, k to be 30, and n to be 10000. Let's examine this code.

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

/*
    matrix: The input matrix
    n: The width and height of matrix
    k: The width and height of a bright spot
    t: The threshold to pass in order to be counted as "bright"
    return value: Number of bright spots.
*/
int brights(float * matrix, int n, int k, float t);

int main () {
    int n = 10000;
    int k = 30;
    float t = 0;
    float * matrix = (float*) malloc(sizeof(float) * n * n);
    clock_t t1, t2;

    for (int i = 0; i < n * n; i++) {
        matrix[i] = 1;
    }
    t1 = clock();
    printf("%d Spots\n", brights(matrix, n, k, t));
    t2 = clock();
    printf("Elapsed Time: %f ms \n",((float)(t2 - t1) / 1000000.0F ) * 1000);
}

int brights(float * matrix, int n, int k, float t) {
    int spots = 0;
    int limit = n - k + 1;

    //Calculate the correct block and grid sizes.
    int vec = n;
    int gan = n;
    if (n > 512) {
        vec = 512;
        gan = ((n * n) / 512) + 1;
    }


    //Data copying once, we don't need it after
    #pragma acc data copyin(matrix[0:n*n])
```

```
45
46      //Outer i loop that sets the block and grid sizes.
47      #pragma acc parallel loop reduction(+:spots) num_gangs(gan) gang vector_length(vec)
48      for(int i = 0; i < limit; i++){
49          #pragma acc loop reduction(+:spots) vector
50          for(int j = 0; j < limit; j++){
51              //boolean to see if current square is a valid square
52              int square = 1;
53              //Iterate over a k x k submatrix
54              for(int l = 0; l < k; l++){
55                  for(int m = 0; m < k; m++){
56                      int this_x = i + l;
57                      int this_y = j + m;
58                      if (matrix[this_x * n + this_y] < t)
59                      square = 0;
60                  }
61              }
62              //Add one to spots if current square is a valid square
63              if (square)
64                  spots = spots + 1;
65          }
66      }
67      return spots;
68  }
```

Let's walk through this program sequentially. Code choices will be explained as we go along. The driver function (main) is included as well. We see the main function on line 14 set all the values as described above, and the timing functions sandwitching the call to brights on line 25.

Algorithmically, we will check all possible squares in a brute-force fashion. However, when using a GPU we are hoping to check every square simultaneously, with each thread checking a k x k square with n x n threads.

Much like manual CUDA programming, we need to determine optimal sizes for the grid and block sizes. We know that at max, a block can have 512 threads. Different from CUDA, we can have as many "blocks" (actually gangs) as we want. The logical thing to do would be to set the vector size (blocks per thread) as high as it can be, and let the number of gangs multiplied by the number of vectors equal the total number of elements.

Next, on line 41, we know that we only need the device to know the contents of the matrix, but reading the data from the device to the host is not necessary. Therefore we schedule the copying to happen and all n * n elements of the matrix are transferred. As a side note, not having the bounds on the matrix variable for any copy operation threw a lot of errors at runtime, mostly "OpenACC runtime error: unspecified launch failure" errors.

We get to the actual kernel work loop here. We want to have absolute control, so rather than using the `kernels` construct, we instead use the `parallel` construct and we set the number of gangs and vector sizes. We also set the program up so that the spots variable will be local and reduced at the end. It is important to note that if you wish to have multiple `acc loop` directives, all of the loops within the scope of the reduced variable must include the reduction (or at least

privatizing the variable) in order to function properly. For this outer loop, we set the number of gangs and vector length to be the numbers as discussed above. Additionally, we put the directive `gang` to indicate that this should be split among the gangs (blocks).

In the inner loop on line 46, we see that we again reduce the spots variable, and this time the `vector` keyword says that we are dividing this loop among the threads of a block.

In order to preserve granularity (each thread is responsible for a k x k square), we decided not to break out of the loop early if there was a non-thresholded element. If the square was indeed a bright spot, we would increment a local variable spots before reducing it.

At the end on line 60, we return the reduced spots. After running the program without OpenACC, and then once with OpenACC, we discovered that the former ran in 45,457 ms while the OpenACC version ran in only 1200 ms, while yielding the same number of correct bright spots! This is a fantastic speedup of 37 x the original speed! We even pulled out the OpenMP version of this program we had written for homework, and this version had run with only 10x the speedup of the original. Overall, with parallelism, we managed to process every potential bright spot in roughly $O(k^2)$ time. This is opposed to the naive version, which would run in $O(k^2n^2)$ time. Overall, this was a huge success!

# B Sample Program 2

Suppose you want to compute the quadratic form in n variables. The quadratic form can be calculated by the formula $q_A = u'Au$ where A is any n x n real symmetric matrix, and u is column vector of length n. This calculation of the quadratic form can be done in parallel as shown below.

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

int parquad(float * a, float * u, int n, float * val) {
    int vec = n;
    int gan = n;
    //Calculate optimal grid and block sizes
    if (n > 512) {
        vec = 512;
        gan = ((n * n) / 512) + 1;
    }

    float sum = 0;

    //We only need to read a and u.
    #pragma acc data copyin(a[0:n*n]) copyin(u[0:n])

    //Split cols among gangs
    #pragma acc parallel loop reduction(+:sum) num_gangs(gan) gang vector_length(vec)
    for (int j = 0; j < n; j++) {
        //split rows among vectors
        #pragma acc loop reduction(+:sum) vector
        for (int i = 0; i < n; i++) {
            //Sum it up using a simplified formula.
            sum += u[i] * a[i * n + j] * u[j];
        }
    }

    //Write the output variable
    *val = sum;
    return 0;
}

int main () {
    int n = 500;

    float * matrix = (float*) malloc(sizeof(float) * n * n);
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            matrix[i*n+j] = matrix[j*n+i] = rand() % 10 + 1;
        }
    }


```

```
46      float * vector = (float*) malloc(sizeof(float) * n);
47      for (int i = 0; i < n; i++) {
48          vector[i] = rand() % 3;
49      }
50
51      float ret;
52      clock_t t1, t2;
53
54      t1 = clock();
55      parquad(matrix, vector, n, &ret);
56      printf("1 x 1: %lf \n", ret);
57      t2 = clock();
58
59      printf("Elapsed Time: %f ms \n",((float)(t2 - t1) / 1000000.0F ) * 1000);
60  }
```

The main function should be pretty self explanatory. We have created an n x n symmetric matrix and a vector of length n (n = 500) both filled with random elements.

Much like Sample Program 1, we find the optimal block and grid size. We want each element to be responsible for the multiplication of three elements. If you look at the multiplication steps involved for computing the quadratic form, you will notice that the resultant number is the product of a vector (formed from $u'A$ multiplied by $u$. If we simplify the formula down, we get that the resultant value is always the sum over all of $i$ and $j$ of u[i] * a[i, j] * u[j].

On line 17, we specify that for the subsequent loop, we only need to read matrices A and u. Then on line 20, we split the loop up so that each gang (block) will be running a single column, while each vector (thread) will be running over a row. We also reduce the entire sum to a variable named sum, which is then written out to an output variable named val.

When not parallelized, the function ran in 980 ms. When parallelized with OpenACC, the function ran in 150 ms. This means that the parallelized version ran with over a 6x speedup compared to the naive version.

# C Sample Program 3

A solid example of OpenACC would be non-negative matrix factorization (NMF). NMF has many real world applications, ranging from image and video rendering to complex physics programs. We chosen to add NMF to this guide since it includes several quintessential parallelization problems, namely matrix multiplication, elementwise matrix operations, and matrix transpose. This solution to NMF is by no means efficient or (to make it more efficient you can combine certain operations as the math to follow will make apparent).

## C.1 Matrix Multiply

```
1  // Multiply M x N by N X K to get a M x K matrix
2  void mult(float *Ret, const float *A, const float *B, int M, int N, int K)
3  {
4      // Copy in from the CPU to the kernel, and copy out
5      #pragma acc parallel loop copyin(A[0:M*N]) copyin(B[0:K*N]) copyout(Ret[0:M*K])
           independent
6      for (int i = 0; i < M; i++) {
7          // The second accelerated loop. Only the outer two loops can be accelerated
8          #pragma acc loop independent
9          for (int j = 0; j < K; j++) {
10             float sum = 0;
11             for (int f = 0 ; f < N; f++) {
12                 sum += A[i * N + f] * B[f * K + j];
13             }
14             Ret[i * K + j] = sum;
15         }
16     }
17 }
```

The matrix mult function uses 3 for loops to carry out the multiplication. In line 5, we use `parallel` to parallelize the section, and `loop` to have the for loop run in parallel. We use `copyin` to transfer A and B to the device, but only transfer back the result using `copyout` to store it in the variable *Ret*. In line 8, we have a second `loop` pragma because the outer two loops can be accelerated. However, the third loop can not be accelerated.

In both pragmas, we are able to include the `independent` construct. This lets the compiler know that the loops are data-independent and can be executed in parallel. We can do this because the inputs are only read from for the calculations, and the output is never used for any calculations.

## C.2 Elementwise Matrix Operations

```
// Element-wise operations. / or * chosen by flag variable, where true is * and false is /
void elementwise(float *A, float *B, int M, int N, int flag)
{
    // Copy in from the CPU to the kernel, and copy out
    float *Ret = (float*)malloc(M * N * sizeof(float));
    #pragma acc parallel loop copyin(A[0:M*N]) copyin(B[0:M*N]) copyout(Ret[0:M*N])
        independent
    for (int i = 0; i < M; i++) {
        // The second accelerated loop. Only the outer two loops can be accelerated
        #pragma acc loop independent
        for (int j = 0; j < N; j++) {
            float result = 0;
            if(flag){result = A[i * N + j] * B[i * N + j];}
            else{result = A[i * N + j] / B[i * N + j];}
            Ret[i * N + j] = result;
        }
    }
    for(int i = 0; i < M * N; i++)
    {
        A[i] = Ret[i];
    }
}
```

The elementwise function is nearly identical to the mult function shown previously. The only difference is that it only uses two for loops and the calculations are different. It again uses `parallel`, `loop`, `copyin`, `copyout`, and `independent`.

## C.3 Matrix Transpose

```
// Stores transposed matrix A in "Ret". Requires the dimensions of A in {M:Rows,N:Columns}
void transpose(float *Ret, float *A, int M, int N){
  //Allocate space, and copy over to Kernel. Then Accelerate loop.
  #pragma acc parallel loop copyin(A[0:M*N]) copyout(Ret[0:M*N]) independent
  for(int i = 0; i < M; i++)
  {
    //Second accelerated loop
    #pragma acc loop independent
    for(int j = 0; j < N; j++)
    {
      Ret[i * M + j] = A[j * N + i];
    }
  }
}
```

The transpose function is again nearly identical to the mult and elementwise functions. It simply has one less parameter, and different operations, but still uses the same parallel constructs.

Together, these three functions provide us the workforce for NMF. In a highly condensed view, NMF takes a $M \times N$ matrix, which in this example we will call $A$, and derive two matrices, $W$ and $H$, which when multiplied together approximate $A$. $W$ and $H$ are calculated as follows:

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

## C.4 NMF, Main

```
#include <stdio.h>
#include <stdlib.h>
// global variables, can be changed as needed
int M = 3;
int N = 3;
int K = 2;
int Niters = 1000;

void mult(float *Ret, const float *A, const float *B, int M, int N, int K);
void transpose(float *Ret, float *A, int M, int N);
void elementwise(float *A, const float *B, int M, int N, int flag);

void nmf(float *w, float *h, float *a, int r, int c, int k, int niters){
    // Fill the array's w and h initially with ones.
    if (r < c) {
        //Intialize variables inside to reduce scope and stack size.
      register int limit = k * c;
      register int lower = k * r;
      for(int t = lower - 1; t >= 0 ; --t)
      {w[t] = 1;h[t] = 1;}
        // initialize the leftover
      for(int t = lower; t < limit; ++t)
        h[t] = 1;
    } else {
      register int limit = k * r;
      register int lower = k * c;
      for(int t = lower - 1; t >= 0 ; --t)
      {w[t] = 1;h[t] = 1;}
      for(int t = lower; t < limit; ++t)
        w[t] = 1;
    }
    // wh memory can be shared as it is used in both equations.
    float* wh = (float*)malloc(r * c * sizeof(float));
    // memory for calculating w.
    float* th = (float*)malloc(k * c * sizeof(float));
    float* Ath = (float*)malloc(r * k * sizeof(float));
    float* whth = (float*)malloc(r * k * sizeof(float));
    // memory for calculating h.
    float* tw = (float*)malloc(k * r * sizeof(float));
```

```
40      float* twA = (float*)malloc(k * c * sizeof(float));
41      float* twwh = (float*)malloc(k * c * sizeof(float));
42      // repeat the calculations for how many iterations
43      for(int iters = 0; iters < niters; iters++)
44      { // Calculating w
45        // Calculate wh
46        mult(wh,w,h,r,c,k);
47        // Calculate h'
48        transpose(th,h,k,c);
49        // Calculate AH'
50        mult(Ath,a,th,r,k,c);
51        // Calculate WHH'
52        mult(whth,wh,th,r,k,c);
53        // Elementwise divide numerator and denominator. store in Ath
54        elementwise(Ath,whth,r,k,1);
55        // Element-wise multiply, store in w.
56        elementwise(w,Ath,r,k,0);
57  // ------------------------------------------------------
58        // calculating h
59        // Calculate wh
60        mult(wh,w,h,r,c,k);
61        // Calculate w'
62        transpose(tw,w,r,k);
63        // Calculate W'A
64        mult(twA,tw,a,k,c,r);
65        // Calculate W'WH
66        mult(twwh,tw,wh,k,c,r);
67        // Elementwise divide numerator and denominator. store in twA
68        elementwise(twA,twwh,k,c,1);
69        // Element-wise multiply, store in h.
70        elementwise(h,Ath,k,c,0);
71      }
72  }
73  int main(){
74      // Set random seed.
75      time_t t; srand((unsigned) time(&t));
76      // Declare memory
77      float* w = (float*)malloc(M * K * sizeof(float));
78      float* h = (float*)malloc(K * N * sizeof(float));
79      float* a = (float*)malloc(M * N * sizeof(float));
80      // populate A. For the purpose of this example, we
81      // will use random numbers less than 50.
82      for(int i = 0; i < M * N; i++){a[i] = rand() % 50;}
83      nmf(w,h,a,M,N,K,Niters);
84      return 0;
85  }
```

To begin looking at this large block of code, let's examine the main function. First, in lines 77-79, we allocate space for the matrices $w$, $h$, and $a$, where the sizes are declared at the top in lines 4-6. In line 82 we initialize $a$ to random values. And lastly we call the nmf function.

In the nmf function, first, in lines 15-31, we are simply initializing all the values in $w$ and $h$ to

1. The slightly convoluted method employed above is simply a way of optimizing this initialization based on the sizes of the matrices.

The next group of lines, 33-41 allocate space for various matrices, some of which are intermediates in the calculation. The variables are named so that if there is a $t$ in front of a $w$, $h$, or $A$, then that variable will have the transpose of that matrix. If there are more than one $w$, $h$, or $A$, in a variable, then the variable will have the result of the matrix multiplication of those matrices. So for example, $Ath$ is $A \times H'$.

Lastly, lines 43-71 do *niters* number of iterations of recalculating $w$ and $h$. They use the intermediates that were declared in 33-41, passing them into the functions shown in the above sections. The calculations being carried out are shown again below:

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

# D Project Responsibilities

Alexander Fu - Setup section, Examples and explanations for the basic OpenACC constructs, Sample Program 1 & explanation, Sample Program 2 & explanation.

David Lin - Overview section, Combining Approaches, Matrix Multiplication, Element-wise Matrix Operations, Matrix Transpose, Main function. General Program 3.

Russell Miller - Proofreading and corrections, Sample Program 2 & 3 explanations. Made sure the tutorial could be followed.