Introduction to OpenACC: Multi-platform Directive-based Parallelism

Yiru Sun, Chen Chen, Hengjiu Kang

9 June 2016[*]

Abstract

This is an OpenACC tutorial for beginners who want to write parallel programs with OpenACC directives. This tutorial is written under the class framework of ECS158 Parallel Programming taught by Professor Matloff in University of California, Davis.

[*]First drafted on 29 May 2016

Contents

List of Tables

Listings

Chapter 1

Introduction

1.1 What Exactly is OpenACC?

OpenACC is a directive-based programming model that brings portability and ease of use to parallel programming. To understand how it works, it is important to have some prior knowledge about OpenMP and CUDA. Long story short, OpenACC is similar to OpenMP in a sense that both use directives to signal compilers what to do with the code. On top of that, OpenACC has added portability so that programs written with OpenACC can be easily ported to work with multiple platforms such as CUDA. This introduction will explain the significance of OpenACC and offer code examples to show how to start writing parallel programs with OpenACC directives.

1.2 Motivations and Significance

Modern image processing and machine learning tasks require more and more computational resources. Computer scientists around the world have been trying to address this ever-growing demand by turning to parallel computing. However, shifting to parallel programming is by no means apparent. While proper implementations of parallel algorithms offer potential performance improvements in tasks such as matrix operations, non-optimized ones could even hinder the performance when compared with their serial counterparts. In order to hide programmers from the hassles of tricky low-level implementations, many parallel libraries and extensions have been developed to help programmers write parallel softwares more easily and promise to deliver reasonable results. Among those libraries, OpenMP is deemed by many as *"the* ***de facto*** *standard for shared-memory programming"*[1]. It is a software implementation that can be deployed on most of the multi-core CPU systems. OpenMP uses pragma directives that are relatively easy to program and can be ignored by compilers that don't understand them. Great convenience is probably one of the many reasons why OpenMP has been so successful, and the use of directives went on to inspire the interface of OpenACC as we will discuss later in detail.

Besides OpenMP, another very important event that has vastly shaped parallel programming and inspired OpenACC is the advent of general-purpose graphics processing unit (GPGPU). GPGPUs are very good at running parallel programs because they are packed with so many processing units in them. Over the years, the companies that design and manufacture GPGPUs have offered programmers with some interfaces and standards to interact with GPGPUs. One example is CUDA which was introduced by NVIDIA in order to interact with CUDA-enabled NVIDIA graphic cards. Despite the great potential of CUDA in performance improvement, many programmers are deterred by the horrendous details and efforts that go into writing a robust and efficient CUDA program. Some called for simpler interfaces which still reap the benefits of the massive parallelism offered by GPGPUs.

In a nutshell, OpenACC was born to inherit the ease of use of directive-based interfaces such as OpenMP, and was designed to be very portable across different platforms such as NVIDIA GPU, AMD Radeon, x86/ARM CPU and Intel Xeon Phi. In the following chapters, we will explore OpenACC directives in greater details by using an introductory matrix multiplication example and two homework problems adapted from ECS158 coursework. It is worth-noting that the code examples in this introduction are in C style.

1.3 What Will Be Covered and What Will Not

This introduction will focus on how to use OpenACC directives to parallelize portions of the code (usually **for loops**), and how to optimize data locality to minimize unnecessary data transfer overheads. This introduction will not cover hardware-specific optimization, interoperability and other advanced features.

Please refer to OpenACC.org[4] for more information.

1.4 Setting up Environment Variables for Omni ompcc Compiler

The compiler of choice of this tutorial is Omni **ompcc** on UC Davis CSIF machines. In order to use ompcc and its backends properly, we need to configure some environment variables first. The following scripts can be written into ~/.login file to run automatically when logging in, or they can be run manually.

```
set path = ($path /home/matloff/Pub)
set path = ($path /home/matloff/Pub/omni/bin)
set path = ($path /usr/lib64/mpich/bin)
set path = ($path /usr/local/cuda-5.5/bin)
set path = ($path /usr/local/cuda-7.5/bin)
setenv LD_LIBRARY_PATH
setenv LD_LIBRARY_PATH /usr/local/cuda-5.5/targets/x86_64-linux/
    lib:$LD_LIBRARY_PATH
setenv LD_LIBRARY_PATH /usr/local/cuda-7.5/targets/x86_64-linux/
    lib:$LD_LIBRARY_PATH
```

Listing 1.1: mmul_acc

To compile OpenACC GPU code with ompcc:
ompcc -o ⟨outputfile name⟩ -acc ⟨input c source code⟩

To compile OpenACC CPU code with ompcc:
ompcc -o ⟨outputfile name⟩ ⟨input c source code⟩

Chapter 2

Matrix Multiplication: Connecting C, OpenMP and CUDA to OpenACC

2.1 Why Matrix Multiplication

In this chapter, we will show you different versions of a simple matrix multiplication example that are written in C, OpenMP and CUDA. Then we will try to link different aspects of C, OpenMP and CUDA to the design of OpenACC. We choose to start with matrix multiplication for it is one of the most important operations in science and engineering, which has sprung many modern applications such as image processing and voice recognition. The examples in this chapter should be relative easy to understand and mainly serve to refresh your memory. The detailed implementations can be skipped.

2.2 Matrix Multiplication in C

We started by writing a matrix multiplication function in C.

```
/*
    Introduction to OpenACC
    mmul_c.c Matrix Multiplication in C
```

```c
    Not optimized
    Not claimed to be efficient
*/
#include "stdio.h"
#include "stdlib.h"

void mmul_c(float* A, float* B, int row, int col, float* out){
    // This code assume that out has been malloc-ed
    int i=0;
    int j=0;
    int inner=0;
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            out[i*col+j] = 0.0;
            for (inner = 0; inner < col; inner++) {
                out[i*col+j] += A[i*col+inner] * B[inner*col+j]*1.0;
            }
        }
    }
}
int main(){
    // crate 3*3 matrix
    float A[]={1,2,3,4,5,6,7,8,9};
    float B[]={2,2,2,2,2,2,2,2,2};
    float out[9];
    mmul_c(A, B, 3, 3, out);
    return 0;
}
```

Listing 2.1: mmul_c

2.3 Matrix Multiplication in OpenMP

Now we can turn the mmul function written in C into a parallel version by using OpenMP directives.

```c
void mmul_omp(float *A, float *B, int row, int col, float *out, ){
    #pragma omp parallel
    {
        int nth = omp_get_thread_num();
        int total = omp_get_num_threads();
        int division = row/total;
        int i=0;
        int j=0;
        int inner = 0;
        #pragma omp for
        for (i = (division*nth); i < (nth==(total-1) ? row : (division*nth+division)); i++) {
            for (j = 0; col < col; j++) {
                out[i*col+j] = 0;
                for (inner = 0; inner < col; inner++) {
                    out[i*col+j] += A[i*col+inner] * B[inner*col+j]*1.0;
                }
            }
        }
    }
}
```

Listing 2.2: mmul_omp

The mmul_omp example above shows usage of OpenMP. Each thread executes individually at the **pragma omp parallel** directive, and the for loop is parallelized by using the **pragma omp for** directive. The designers of OpenACC adopted the concept of directives and they came up with their own versions of the similar directives in OpenACC.

2.4 Matrix Multiplication in CUDA

CUDA is parallel platform that interfaces with NVIDIA GPUs. Matrix Multiplication can also be performed on a GPU as shown below.

```

__global__ void kernel(float *A, float *B, int row, int col,
     float *out){
    int nth = threadIdx.x;
    int division = r1/total;
    int i=0;
    int j=0;
    int inner=0;
    for (i = (division*nth); i < (nth==(total-1) ? row : (
     division*nth+division)); i++) {
        for (j = 0; j < col; j++) {
            out[i*col+j] = 0;
            for (inner = 0; inner < col; inner++) {
                out[i*col+j] += a[i*col+inner] * b[inner*col+j
     ]*1.0;
            }
        }
    }
}

void mmul_cuda(float *A, float *B, int row, int col, float *out){
    float *dA, *dB, *dout;
    int msize = row*col*sizeof(float);
    cudaMalloc((void**)&dA, msize);
    cudaMalloc((void**)&dB, msize);
    cudaMalloc((void**)&dm, msize);
    cudaMemcpy(dA, A, msize, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, B, msize, cudaMemcpyHostToDevice);

    int blockSize = 4;  // use 4 blocks
    dim3 dimGrid(blockSize, 1);
    dim3 dimBlock(1,1,1);

    kernel<<<dimGrid, dimBlock>>>(dA, dB, row, col, dm);

    cudaMemcpy(out, dm, msize, cudaMemcpyDeviceToHost);
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dm);

    return;

}
```

Listing 2.3: mmul_cuda

Two key features of CUDA inspired OpenACC design are the explicit allocation of memory and transfer of data. With CUDA, programmers have decent control over data flow and memory allocation, which enables them to optimize data locality. OpenACC designers also want programmers to have some degree of control over data and memory, so they incorporate some special directives and structures dedicated to the purpose of data locality optimization. By reducing the overheads of data movement, a parallel program can see great performance improvement.

Another important design detail of CUDA is the separation between host memory space and device memory space. Device memory space can be thought of as a programmer-managed cache which is crucial to the performance of a GPU program. The unnecessary transfer of data between host and device often causes huge performance problem and OpenACC offers some directives to control and manage the transfer of data between them.

Up until this point, chapter 2 has established connections between C, OpenMP, CUDA and OpenACC. Chapter 3 will introduce OpenACC in greater details.

Chapter 3

Introduction to OpenACC

3.1 Available Compilers

OpenACC offers a set of directives to suggest compilers what to do with the code, but the backend implementation depends on the compiler and flags being used. The list below shows several OpenACC compatible compilers.

Name	Developer	Commercial	Backend	Platform
Omni	Omni Compiler Project	NO	CUDA	X86 Only
accULL	High Performance Computing Group	NO	CUDA/OpenCL	ARM/X86
GNU-Openacc	GNU project	NO	CUDA/OpenCL	Not complete
PGI	PGI	YES	CUDA/OpenCL	X86 Only

Table 3.1: OpenACC compilers

As mentioned in the introductory chapter, we are using Omni ompcc compiler in this tutorial.

3.2 Basic Directives

Just like OpenMP, OpenACC uses directives to communicate with compilers. Below is a list of basic directives that are useful in starting off with OpenACC.

1. **#pragma acc kernels**
   kernels construct is probably the easiest one to work with. It can be placed before a for loop to signal the compiler that the for loop immediately after **kernels** directive might be parallelized. That being said, it is up to the compiler to decide whether the parallelism is going to happen and the outcome is not guaranteed. **kernels** is suitable for inexperienced programmers and cases where the loops are too complicated to analyze.

2. **#pragma acc parallel**
   parallel construct is similar to **#pragma acc kernels**, but it goes further and tells the compiler explicitly that the for loop immediately below can be parallelized. The programmer is responsible for making sure that the loop is safe to parallelize. It is worth noting that **parallel** needs to be used with another directive to properly function.

3. **#pragma acc loop [options]**
   loop directive is usually used together with **parallel** directive in the form of **#pragma acc parallel loop** in order to signal the compiler to parallelize the loop immediately after the resultant directive.

4. **#pragma acc data [options]**
   data construct of OpenACC precedes a special set of clauses for data transfer and memory allocation. The options of **data** directive are similar to a combination of CUDA functions working collectively. For example, the **copy** clause can be used to form **#pragma acc data copy(A)** and the resultant directive is similar to the following CUDA pseudocode:

```
cudaMalloc(A_device)  // Allocate Memory on Device
cudaMemcpy(A_host, A_device, cudaMemcpyHostToDevice)  //
     Initialize A with the Data from Host
cudaMemcpy(A_host, A_device, cudaMemcpyDeviceToHost)  //
     Copy the Resultant Data back to Host
cudaFree(A_device)  // Free Device Memory
```

Listing 3.1: data_copy_CUDA

3.3 Matrix Multiplication in OpenACC

We can now implement matrix multiplication in OpenACC. The following code is based on the C version of matrix multiplication in Chapter 2.

```
void mmul_acc(float* A, float* B, int row, int col, float* out){
    // This code assume that out has been malloc-ed
    int i=0;
    int j=0;
    int inner=0;
    float temp_sum=0;
    #pragma acc data create(A[0:(row*col)],B[0:(row*col)]) copy(
     out[0:row*col])
    {
        #pragma acc parallel loop
        for (i = 0; i < row; i++) {
            #pragma acc loop
            for (j = 0; j < col; j++) {
                out[i*col+j] = 0.0;
                #pragma acc loop reduction(+:temp_sum)
                for (inner = 0; inner < col; inner++) {
                    temp_sum = A[i*col+inner] * B[inner*col+j
     ]*1.0;
                }
                out[i*col+j]=temp_sum;
                temp_sum=0;
            }
        }
    }
}
```

Listing 3.2: mmul_acc

In this example, we use the following directives to implement parallelism:

- #pragma acc data create(A, B)
  A and B are two input matrices and will not be retrieved from the device after the computations are finished. So it is safe to use **data create** construct instead of **data copy** to create dedicated memory space for A and B in the device but do not initialize or copy back to host after the job has been done. This move is usually very effective in improving data locality and reduce communication overhead.

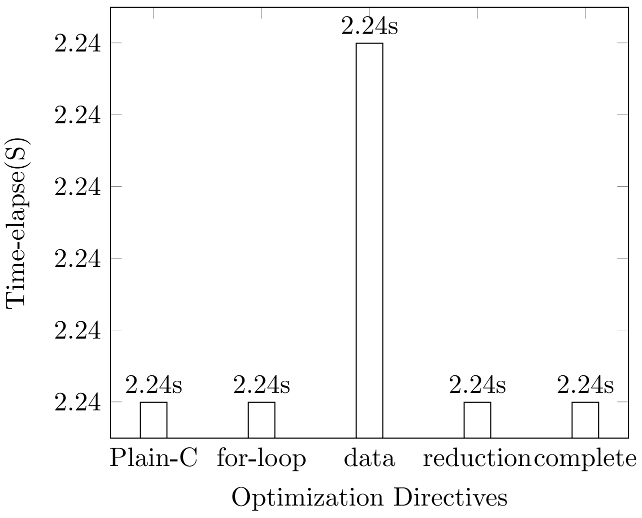
- #pragma acc parallel loop
  This directive enables loop optimization and parallelism as we mentioned in point 3 of 3.2.

- #pragma acc loop
  Inside a nested for-loop, we need to use **#pragma acc parallel loop** first right before the outermost for-loop, then use **#pragma acc loop** in the inner for-loop to signal further parallelization.

- reduction(+:temp_sum)
  reduction is a special directive in OpenACC. It can monitor the value of a certain variable (in this case: temp_sum), perform a certain function (in this case: +) to the variable and reduce the results of all the operations into one single result. One of the following basic functions can act as a reduction function: **+, *, min, max**.

Performance comparison in directives

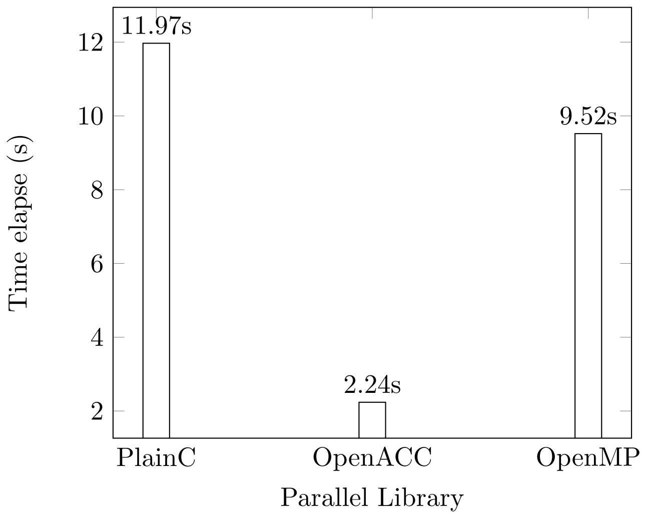
Also, we need to be aware that, different directives have different optimization performance, and the performance depends on each application. For example, we want **for loop** optimization and **reduce** optimization for dot multiplication summation. In other application, like path-finding algorithm, we may want to create on-device memory space for faster variable access, because path-finding is memory accessing massive algorithm.
In this section, we compare the improvement of each directive to original program step by step. When we are using GCC to compile plain-C code without any flag, the execution time is about 12 seconds.



We found that the differences between directives are not significant, which is out of expectation. An possible explanation is that this Matrix multiplication is an typical application that ompcc can easily maximize the optimization. So it is reasonable that ompcc compiled plain-C code is already 5 times faster than gcc compiler, and user added customized directives will be ignored or have the same effect.

Performance Comparison between Difference Libraries

Our test program performs a 50*50 matrix multiplication for 30000 times, and the following time elapse data were recorded from running the command: time mmul_**x**.out, where **x** refers to c, acc or omp



It is easily observed that the performance improvement by using OpenACC directives is enormous in the case of matrix multiplication, partly because it is embarrassingly parallel and therefore benefit largely from the use of parallel computing.

Chapter 3 has covered most of the OpenACC directives in detail. The following two chapters will introduce some code examples to show how OpenACC can be applied in various applications.

Chapter 4

OpenACC Example 1: Path Finder

This chapter is all about OpenACC in action and a new **atomic update** directive for critical region access. We use OpenACC to parallelize a path finder program originally written in C.

The path finder program takes an adjacency matrix and find all possible paths of length **k** that are connected, starting from a given vertex. Repeated vertices are allowed.

For example, if **k** is 5 and row i of paths is 99 2 0 3 8 88
This is the path 99 $\rightarrow$ 2 $\rightarrow$ 0 $\rightarrow$ 3 $\rightarrow$ 8 $\rightarrow$ 88
Loops are allowed, e.g. 5 $\rightarrow$ 2 $\rightarrow$ 0 $\rightarrow$ 2 $\rightarrow$ 0 $\rightarrow$ 88

Here is a list of arguments taken by the path finder function:
adjm: Adjacency matrix of a directed graph.
n: Number of vertices in the graph.
k: Length of each path.
paths: Output matrix of paths.
numpaths: Number of paths.

The output matrix: **paths**, will have **numpaths** rows and **k**+1 columns, and likely to have unused rows at the bottom. The space will be pre-allocated in the main function.

Here is the C code to implement the path finder:

```c
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <stdlib.h>

double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL)) return 0;
    return (double)time.tv_sec + (double)time.tv_usec * .000001;
}

void findpaths(int *adjm, int n, int k, int *paths, int *numpaths
    ) {
    int count = 0;
    int times = n;
    for (int i = 0; i < k; ++i) {times = times * n;}  //
     Dimension of time = n^k

    int* arr = (int*) malloc (k+1); // Create an array arr that
     is indexed from 0 to k

    for (int i = 0; i < k+1; ++i) {arr[i] = 0;} // Initialize all
      elements in arr to be 0
    for (int i = 0; i < times; ++i)
    {
        int temp = i, index = k, check = 1, r;
        while(temp!=0)
        {
            r=(temp%n);
            temp/=n;
            arr[index] = r;
            index--;
        }

        for (int i = 0; i < k; ++i)
        {
            check *= adjm[arr[i]*n+arr[i+1]];
        }

        if(check)
        {
            for (int i = 0; i < k+1; ++i)
            {
                paths[count*(k+1)+i] = arr[i];
            }
      count++;
        }
    }
    *numpaths = count;
}

int main(int argc, char *argv[])
{
  int n = 5, k = 4;
  int path[2000000];
  int numpaths;
  int loopitr;
54
55   static int A[400] = {
56     0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0,
57     1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1,
58     1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0,
59     1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0,
60     1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0,
61     1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1,
62     1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
63     0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
64     0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1,
65     1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1,
66     0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
67     1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1,
68     0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0,
69     0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0,
70     0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0,
71     1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1,
72     0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
73     0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1,
74     1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0,
75     0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0};
76
77   double time_begin=get_wall_time();
78   for (loopitr = 0; loopitr < 30000; loopitr++) {
79     findpaths(A, n, k, path, &numpaths);
80   }
81   double time_end=get_wall_time();
82   printf("Total elapsed time is: %.4lf\n", (time_end - time_begin
       ));
83
84   printf("%d\n",numpaths);
85   for (int i = 0; i < numpaths; ++i)
86   {
87     for (int j = 0; j < k+1; ++j)
88     {
89       printf("%d->", path[i*(k+1)+j]);
90     }
91     printf("\n");
92   }
93
94   return 0;
95 }
```

Listing 4.1: path

To look for paths with length of 4, each path would have 5 vertices. Since loop is allowed and we have a total of n vertexes, we would check all possible paths from $0 \to 0 \to 0 \to 0 \to 0$ to $n \to n \to n \to n \to n$.

The total run times would be n to the power of 5 in line 20. Let's give an order to these vertexes: $i \to j \to k \to l \to m$. We need to check each path in the matrix to make sure the whole path is correct. If $m_{i,j}$ equals 1, there is a path from i to j. Thus, we would check $m_{i,j}$, $m_{j,k}$, $m_{k,l}$ and $m_{l,m}$ in line 31. If the path exists, we count it and put it into the output path matrix.

Here is the modified OpenACC version with parallelism.

```
 1 #include <stdio.h>
 2 #include <time.h>
 3 
 4 
 5 void findpaths(int *adjm, int n, int k, int *paths, int *numpaths ) {
 6     int count = 0;
 7     int times = n;
 8     for (int i = 0; i < k; ++i) {times = times * n;}  // Dimension of time = n^k
 9     int arr[k+1];     // Create an array arr that is indexed from 0 to k
10 
11     for (int i = 0; i < k+1; ++i) {arr[i] = 0;} // Initialize all elements in arr to be 0
12     #pragma acc parallel loop
13     for (int i = 0; i < times; ++i)
14     {
15         int temp = i, index = k, check = 1, r;
16         while(temp!=0)
17         {
18             r=(temp%n);
19             temp/=n;
20             arr[index] = r;
21             index--;
22         }
23 
24         for (int i = 0; i < k; ++i)
25         {
26             check *= adjm[arr[i]*n+arr[i+1]];
27         }
28 
29         if(check)
30         {
31             for (int i = 0; i < k+1; ++i)
32             {
33                 paths[count*(k+1)+i] = arr[i];
34             }
35             #pragma acc atomic update
36       count++;
37         }
38     }
39 
40 
41     *numpaths = count;
42 }
43 
44 int main(int argc, char const *argv[])
45 {
46   int n = 5, k = 4;
47   int path[2000000], numpaths;
48   static int A[400] = {
49     0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0,
50     1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1,
51      1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0,
52      1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0,
53      1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0,
54      1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1,
55      1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
56      0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
57      0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1,
58      1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1,
59      0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
60      1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1,
61      0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0,
62      0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0,
63      0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0,
64      1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1,
65      0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
66      0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1,
67      1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0,
68      0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0};
69
70    findpaths(A, n, k, path, &numpaths);
71
72    printf("%d\n",numpaths);
73    for (int i = 0; i < numpaths; ++i)
74    {
75      for (int j = 0; j < k+1; ++j)
76      {
77        printf("%d->", path[i*(k+1)+j]);
78      }
79      printf("\n");
80    }
81
82
83    return 0;
84  }
```

Listing 4.2: path_acc

The OpenACC version of the path finder code is pretty standard and tested to render about twice the speed compared with its serial counterpart. Our code is by no means efficient and more parallelism and data locality optimization can be done. The purpose of this example is to point out a very important aspect of parallel programming: critical region access and synchronization.

#pragma acc atomic update directive in line 44 signals to the compiler that the line immediately after is a critical region access. This is crucial especially in shared-memory implementation due to potential race conditions.

Chapter 5

OpenACC Example 2: Bright Spot Counter

Consider an n x n matrix of image pixels, with brightness values in the range of [0,1]. A bright spot of size k and threshhold b is defined as a k x k subimage, with the pixels being contiguous and with each one having brightness at least b. Here is a C program that returns a count of all bright spots.

Here is a list of arguments taken by the bright spot counter function:
n is matrix dimension
k is subimage dimension
start and end are for parallelism
pix is the pointer to the matrix.

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MATRIX_MAX 1
#define MAX_SIZE 1400
#define N_ITERS 1000


void seed(int r, int c, float *out) {    //generate a random
     matrix
    time_t t;
    srand((unsigned) time(&t));
    for (int i=0;i<r;i++){
        for (int j=0;j<c;j++){
            out[i*c+j]=(float)(MATRIX_MAX*((int)(rand()%100)/100.0));
        }
    }
}


    // n = matrix dimension; k = subimage dimension; start and end are for parallelism; pix is the pointer to the matrix.
int brightssplit (float *pix, int n, int k, float thresh, int start, int end) {
  float count = 0, check[n];
  for (int i = 0; i < n; ++i)
  { // iterate throught ALL n rows
    // Clear book keeping variables for a new round
          float minus, sum = 0;
    // Count the number of bright pixel in a single row of 1 x k
    // i * n locate row, + j locate column
      for (int j = start; j < start+k; ++j) if (pix[i*n+j] > thresh) {sum++;}
    // end - k + 1 basically recovers the boundary by the 1st column index of each window
    // which could be understood as the window column index
    // instead of the last column index which is end itself
    // Now we run from the starting window column index to the end WCI
    // Visually, we are moving a 1 x k block across row i to track bright block
      for (int j = start; j < (end-k+1); ++j)
      {
    // minus:
    // This is a smart way of moving the 1 x k block to the right.
    // Instead of using excessive for loops and repetitive checking,
    // we simply look at the left most pixel of that block,
    // and later we can see what we could do with sum.
        if (pix[i*n+j] > thresh) {minus = 1;} else {minus = 0;}
    // Check List for keeping track of the number of 1 x k bright unit
    // in terms of window column index
    // Initialized when going through the first row

        if (i == 0) {if (sum >= k) {check[j] = 1;} else {check[j] = 0;}}
    // Now we are out of the first row, that means we get to +/- to the list,
    // and hopefully hit a bring window when count >= k!
    // Keep Count++ when check[j] >= k
    // Clear it when we hit a 1 x k block with non-bright pixels.
        else {
          if (sum >= k) {
            check[j] ++;
            if (check[j] >= k) {count ++;}}
          else {check[j] = 0;}}
    // Now we are about to move the 1 x k block to the right. Rmb the minus?
    // We can now use it to change sum, together with the information of
    // the one new pixel the 1 x k block now includes.
```

```c
60      // Again, elegant way of avoiding repetitive checking!
61
62          if(minus == 0 && pix[i*n+j+k] > thresh) {sum++;}
63          else if(minus == 1 && pix[i*n+j+k] <= thresh) {sum--;}
64        }
65    }
66
67    return count;
68 }
69
70 int brights (float *pix, int n, int k, float thresh) {
71      brightssplit (pix, n, k, thresh, 0, n);
72 }
73
74
75 int main (int argc, char const *argv[])
76 {
77
78    float A[MAX_SIZE*MAX_SIZE]; // use one dimensional array to represent the matrix
79    seed(MAX_SIZE,MAX_SIZE,A);  // generate random matrix
80      int count = 0;
81      int i=0;
82      for (i=0;i<N_ITERS;i++){
83          count += brights (A, 1000, 5, 0.2);
84      }
85
86      printf("There are average %d starts in %d iters.\n", count/N_ITERS, N_ITERS);
87    return 0;
88 }
```

Listing 5.1: Brightcounter

We run the function brights() for 1000 times to get a significantly high run time for the sake time analysis. In the function brightssplit(), the program searches the matrix row by row with the for loop in line 23. Then it checks if there are **k** numbers of continuous bright pixels from line 29 and store the information in the first row of **pix** matrix. When the program checks each row, it adds the information to the first row of **pix** matrix. Finally, the program calculates the data in the first row of **pix** matrix and outputs the number of bright spots.

Here is the modified OpenACC version with parallelism.

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MATRIX_MAX 1
#define MAX_SIZE 1400
#define N_ITERS 1000
```

```


void seed(int r, int c, float *out) {   //generate a random
     matrix
    time_t t;
    srand((unsigned) time(&t));
    for (int i=0;i<r;i++){
        for (int j=0;j<c;j++){
            out[i*c+j]=(float)(MATRIX_MAX*((int)(rand()%100)
     /100.0));
        }
    }
}


#pragma acc routine seq
int brightssplit (float *pix, int n, int k, float thresh, int
     start, int end) {
    float count = 0, check[n];
    #pragma acc data copy(pix[0:n*n], check[0:n])
    {

        #pragma acc loop
        for (int i = 0; i < n; ++i)
        {
            float minus, sum = 0;

            for (int j = start; j < start+k; ++j){
                if (pix[i*n+j] > thresh){
                    sum++;
                }
            }
            #pragma acc loop
            for (int j = start; j < (end-k+1); ++j)
            {
                if (pix[i*n+j] > thresh){
                    minus = 1;
                } else {
                    minus = 0;
                }

                if (i == 0) {
                    if (sum >= k) {
                        check[j] = 1;
                    } else {
                        check[j] = 0;
                    }
                }
                else {
                    if (sum >= k) {
                        check[j] ++;
                        if (check[j] >= k) {
                            count ++;
                        }
                    }
                    else {
                        check[j] = 0;
                    }
                }

                if(minus == 0 && pix[i*n+j+k] > thresh) {sum++;}
                else if(minus == 1 && pix[i*n+j+k] <= thresh) {
```

```
     sum--;}
66                  }
67              }
68      }
69  
70      return count;
71  }
72  
73  int brights (float *pix, int n, int k, float thresh) {
74    int i=0;
75      int blocks = n/k-1;
76      int count = 0;
77    #pragma acc data create(pix)
78      #pragma acc parallel loop reduction(+:count)
79      for (i=0;i<blocks;i++){
80          count += brightssplit (pix, n, k, thresh, k*i, k*(i+1)+k
     -1);
81      }
82  
83      count += brightssplit(pix, n, k, thresh, k*i, n);
84      return count;
85  }
86  
87  
88  int main (int argc, char const *argv[])
89  {
90  
91    float A[MAX_SIZE*MAX_SIZE];
92      seed(MAX_SIZE,MAX_SIZE,A);
93  
94      int count = 0;
95      int i=0;
96      for (i=0;i<N_ITERS;i++){
97          count += brights (A, 1000, 5, 0.2);
98      }
99  
100     printf("There are average %d starts in %d iters.\n", count/
     N_ITERS, N_ITERS);
101   return 0;
102 }
```

Listing 5.2: Brightcounter_acc

The OpenACC directives in this bright spot finder program have been introduced in greater details in Chapter 3. However, it is worth noting that many constructs can be stacked together to achieve a more compact code, just like **#pragma acc parallel loop reduction(+:count)** in line 78.

Appendix A

Contribution

Thanks to each of the group members, Chen chen, Yiru Sun and Hengjiu Kang.
Chen chen mainly worked on Chapter 1 and Chapter 2, Yiru Sun worked on Chapter 4 and Chapter 5. Hengjiu Kang worked on source code porting and Chapter 3.
We started the first draft since May.29 2016, and we were working together through the Final weeks. Finally we finished the tutorial on June.9 2016.

Appendix B

Contact information

Chen Chen: cccche@ucdavis.edu
Yiru Sun: yrsun@ucdavis.edu
Hengjiu Kang: hjkang@ucdavis.edu `khjtony.github.io`

Appendix C

Source code

```
set path = ($path /home/matloff/Pub)
set path = ($path /home/matloff/Pub/omni/bin)
set path = ($path /usr/lib64/mpich/bin)
set path = ($path /usr/local/cuda-5.5/bin)
set path = ($path /usr/local/cuda-7.5/bin)
setenv LD_LIBRARY_PATH
setenv LD_LIBRARY_PATH /usr/local/cuda-5.5/targets/x86_64-linux/
    lib:$LD_LIBRARY_PATH
setenv LD_LIBRARY_PATH /usr/local/cuda-7.5/targets/x86_64-linux/
    lib:$LD_LIBRARY_PATH
```

Listing C.1: mmul_acc

```
/*
    Introduction to OpenACC
    mmul_c.c Matrix Multiplication in C
    Not optimized
    Not claimed to be efficient
*/
#include "stdio.h"
#include "stdlib.h"

void mmul_c(float* A, float* B, int row, int col, float* out){
    // This code assume that out has been malloc-ed
    int i=0;
    int j=0;
    int inner=0;
    for (i = 0; i < row; i++) {
        for (j = 0; j < col; j++) {
            out[i*col+j] = 0.0;
            for (inner = 0; inner < col; inner++) {
                out[i*col+j] += A[i*col+inner] * B[inner*col+j
    ]*1.0;
            }
        }
    }
}
```

```c
int main(){
    // crate 3*3 matrix
    float A[]={1,2,3,4,5,6,7,8,9};
    float B[]={2,2,2,2,2,2,2,2,2};
    float out[9];
    mmul_c(A, B, 3, 3, out);
    return 0;
}
```

Listing C.2: mmul_c

```c
void mmul_omp(float *A, float *B, int row, int col, float *out, ) {
    #pragma omp parallel
    {
        int nth = omp_get_thread_num();
        int total = omp_get_num_threads();
        int division = row/total;
        int i=0;
        int j=0;
        int inner = 0;
        #pragma omp for
        for (i = (division*nth); i < (nth==(total-1) ? row : (division*nth+division)); i++) {
            for (j = 0; col < col; j++) {
                out[i*col+j] = 0;
                for (inner = 0; inner < col; inner++) {
                    out[i*col+j] += A[i*col+inner] * B[inner*col+j]*1.0;
                }
            }
        }
    }
}
```

Listing C.3: mmul_omp

```c

__global__ void kernel(float *A, float *B, int row, int col, float *out){
    int nth = threadIdx.x;
    int division = r1/total;
    int i=0;
    int j=0;
    int inner=0;
    for (i = (division*nth); i < (nth==(total-1) ? row : (division*nth+division)); i++) {
        for (j = 0; j < col; j++) {
            out[i*col+j] = 0;
            for (inner = 0; inner < col; inner++) {
                out[i*col+j] += a[i*col+inner] * b[inner*col+j]*1.0;
            }
        }
    }
}

void mmul_cuda(float *A, float *B, int row, int col, float *out){
    float *dA, *dB, *dout;
    int msize = row*col*sizeof(float);
    cudaMalloc((void**)&dA, msize);
    cudaMalloc((void**)&dB, msize);
    cudaMalloc((void**)&dm, msize);
    cudaMemcpy(dA, A, msize, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, B, msize, cudaMemcpyHostToDevice);

    int blockSize = 4;  // use 4 blocks
    dim3 dimGrid(blockSize, 1);
    dim3 dimBlock(1,1,1);

    kernel<<<dimGrid, dimBlock>>>(dA, dB, row, col, dm);

    cudaMemcpy(out, dm, msize, cudaMemcpyDeviceToHost);
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dm);

    return;

}
```

Listing C.4: mmul_cuda

```
cudaMalloc(A_device)  // Allocate Memory on Device
cudaMemcpy(A_host, A_device, cudaMemcpyHostToDevice)  // Initialize A with the Data from Host
cudaMemcpy(A_host, A_device, cudaMemcpyDeviceToHost)  // Copy the Resultant Data back to Host
cudaFree(A_device)  // Free Device Memory
```

Listing C.5: data_copy_CUDA

```
void mmul_acc(float* A, float* B, int row, int col, float* out){
    // This code assume that out has been malloc-ed
    int i=0;
    int j=0;
    int inner=0;
    float temp_sum=0;
    #pragma acc data create(A[0:(row*col)],B[0:(row*col)]) copy(out[0:row*col])
    {
        #pragma acc parallel loop
        for (i = 0; i < row; i++) {
            #pragma acc loop
            for (j = 0; j < col; j++) {
                out[i*col+j] = 0.0;
                #pragma acc loop reduction(+:temp_sum)
                for (inner = 0; inner < col; inner++) {
                    temp_sum = A[i*col+inner] * B[inner*col+j]*1.0;
                }
                out[i*col+j]=temp_sum;
                temp_sum=0;
            }
        }
    }
}
```

Listing C.6: mmul_acc

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

double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL)) return 0;
    return (double)time.tv_sec + (double)time.tv_usec * .000001;
}

void findpaths(int *adjm, int n, int k, int *paths, int *numpaths
    ) {
    int count = 0;
    int times = n;
    for (int i = 0; i < k; ++i) {times = times * n;}  //
    Dimension of time = n^k

    int* arr = (int*) malloc (k+1); // Create an array arr that
    is indexed from 0 to k

    for (int i = 0; i < k+1; ++i) {arr[i] = 0;} // Initialize all
      elements in arr to be 0
    for (int i = 0; i < times; ++i)
    {
        int temp = i, index = k, check = 1, r;
        while(temp!=0)
        {
            r=(temp%n);
            temp/=n;
            arr[index] = r;
            index--;
        }

        for (int i = 0; i < k; ++i)
        {
            check *= adjm[arr[i]*n+arr[i+1]];
        }

        if(check)
        {
            for (int i = 0; i < k+1; ++i)
            {
                paths[count*(k+1)+i] = arr[i];
            }
      count++;
        }
    }
    *numpaths = count;
}

int main(int argc, char *argv[])
{
  int n = 5, k = 4;
  int path[2000000];
  int numpaths;
  int loopitr;

  static int A[400] = {
    0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0,
    1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1,
    1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0,
    1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0,
    1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1,
    1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
    0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1,
    1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1,
    0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
    1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1,
    0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0,
    0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0,
    0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0,
    1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1,
    0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1,
    1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0,
    0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0};

  double time_begin=get_wall_time();
  for (loopitr = 0; loopitr < 30000; loopitr++) {
    findpaths(A, n, k, path, &numpaths);
  }
  double time_end=get_wall_time();
  printf("Total elapsed time is: %.4lf\n", (time_end - time_begin));

  printf("%d\n",numpaths);
  for (int i = 0; i < numpaths; ++i)
  {
    for (int j = 0; j < k+1; ++j)
    {
      printf("%d->", path[i*(k+1)+j]);
    }
    printf("\n");
  }

  return 0;
}
```

Listing C.7: path

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


void findpaths(int *adjm, int n, int k, int *paths, int *numpaths) {
    int count = 0;
    int times = n;
    for (int i = 0; i < k; ++i) {times = times * n;}  // Dimension of time = n^k
    int arr[k+1];     // Create an array arr that is indexed from 0 to k

    for (int i = 0; i < k+1; ++i) {arr[i] = 0;} // Initialize all elements in arr to be 0
   #pragma acc parallel loop
    for (int i = 0; i < times; ++i)
    {
        int temp = i, index = k, check = 1, r;
        while(temp!=0)
```

```c
        {
            r=(temp%n);
            temp/=n;
            arr[index] = r;
            index--;
        }

        for (int i = 0; i < k; ++i)
        {
            check *= adjm[arr[i]*n+arr[i+1]];
        }

        if(check)
        {
            for (int i = 0; i < k+1; ++i)
            {
                paths[count*(k+1)+i] = arr[i];
            }
            #pragma acc atomic update
      count++;
        }
    }


    *numpaths = count;
}

int main(int argc, char const *argv[])
{
  int n = 5, k = 4;
  int path[2000000], numpaths;
  static int A[400] = {
    0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0,
    1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1,
    1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0,
    1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0,
    1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0,
    1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1,
    1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0,
    0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1,
    1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1,
    0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
    1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1,
    0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0,
    0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0,
    0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0,
    1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1,
    0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1,
    1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0,
    0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0};

  findpaths(A, n, k, path, &numpaths);

  printf("%d\n",numpaths);
  for (int i = 0; i < numpaths; ++i)
  {
    for (int j = 0; j < k+1; ++j)
    {
      printf("%d->", path[i*(k+1)+j]);
    }
    printf("\n");
  }


  return 0;
}
```

Listing C.8: path_acc

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MATRIX_MAX 1
#define MAX_SIZE 1400
#define N_ITERS 1000


void seed(int r, int c, float *out) {   //generate a random matrix
    time_t t;
    srand((unsigned) time(&t));
    for (int i=0;i<r;i++){
        for (int j=0;j<c;j++){
            out[i*c+j]=(float)(MATRIX_MAX*((int)(rand()%100)/100.0));
        }
    }
}


    // n = matrix dimension; k = subimage dimension; start and end are for parallelism; pix is the pointer to the matrix.
int brightssplit (float *pix, int n, int k, float thresh, int start, int end) {
  float count = 0, check[n];
  for (int i = 0; i < n; ++i)
  { // iterate throught ALL n rows
    // Clear book keeping variables for a new round
          float minus, sum = 0;
    // Count the number of bright pixel in a single row of 1 x k
    // i * n locate row, + j locate column
      for (int j = start; j < start+k; ++j) if (pix[i*n+j] > thresh) {sum++;}
    // end - k + 1 basically recovers the boundary by the 1st column index of each window
    // which could be understood as the window column index
    // instead of the last column index which is end itself
    // Now we run from the starting window column index to the end WCI
    // Visually, we are moving a 1 x k block across row i to track bright block
      for (int j = start; j < (end-k+1); ++j)
      {
    // minus:
    // This is a smart way of moving the 1 x k block to the right.
    // Instead of using excessive for loops and repetitive checking,
    // we simply look at the left most pixel of that block,
    // and later we can see what we could do with sum.
        if (pix[i*n+j] > thresh) {minus = 1;} else {minus = 0;}
    // Check List for keeping track of the number of 1 x k bright unit
    // in terms of window column index
    // Initialized when going through the first row

        if (i == 0) {if (sum >= k) {check[j] = 1;} else {check[j] = 0;}}
    // Now we are out of the first row, that means we get to +/- to the list,
    // and hopefully hit a bring window when count >= k!
    // Keep Count++ when check[j] >= k
    // Clear it when we hit a 1 x k block with non-bright pixels.
        else {
          if (sum >= k) {
            check[j] ++;
            if (check[j] >= k) {count ++;}}
          else {check[j] = 0;}}
    // Now we are about to move the 1 x k block to the right. Rmb the minus?
    // We can now use it to change sum, together with the information of
    // the one new pixel the 1 x k block now includes.
    // Again, elegant way of avoiding repetitive checking!

        if(minus == 0 && pix[i*n+j+k] > thresh) {sum++;}
        else if(minus == 1 && pix[i*n+j+k] <= thresh) {sum--;}
      }
  }

  return count;
}

int brights (float *pix, int n, int k, float thresh) {
    brightssplit (pix, n, k, thresh, 0, n);
}


int main (int argc, char const *argv[])
{

  float A[MAX_SIZE*MAX_SIZE]; // use one dimensional array to represent the matrix
  seed(MAX_SIZE,MAX_SIZE,A);  // generate random matrix
    int count = 0;
    int i=0;
    for (i=0;i<N_ITERS;i++){
        count += brights (A, 1000, 5, 0.2);
    }

    printf("There are average %d starts in %d iters.\n", count/N_ITERS, N_ITERS);
  return 0;
}
```

Listing C.9: Brightcounter

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MATRIX_MAX 1
#define MAX_SIZE 1400
```

```
#define N_ITERS 1000


void seed(int r, int c, float *out) {    //generate a random matrix
    time_t t;
    srand((unsigned) time(&t));
    for (int i=0;i<r;i++){
        for (int j=0;j<c;j++){
            out[i*c+j]=(float)(MATRIX_MAX*((int)(rand()%100)/100.0));
        }
    }
}


#pragma acc routine seq
int brightssplit (float *pix, int n, int k, float thresh, int start, int end) {
    float count = 0, check[n];
    #pragma acc data copy(pix[0:n*n], check[0:n])
    {

        #pragma acc loop
        for (int i = 0; i < n; ++i)
        {
            float minus, sum = 0;

            for (int j = start; j < start+k; ++j){
                if (pix[i*n+j] > thresh){
                    sum++;
                }
            }
            #pragma acc loop
            for (int j = start; j < (end-k+1); ++j)
            {
                if (pix[i*n+j] > thresh){
                    minus = 1;
                } else {
                    minus = 0;
                }

                if (i == 0) {
                    if (sum >= k) {
                        check[j] = 1;
                    } else {
                        check[j] = 0;
                    }
                }
                else {
                    if (sum >= k) {
                        check[j] ++;
                        if (check[j] >= k) {
                            count ++;
                        }
                    }
                    else {
                        check[j] = 0;
                    }
                }

                if(minus == 0 && pix[i*n+j+k] > thresh) {sum++;}
                else if(minus == 1 && pix[i*n+j+k] <= thresh) {
    sum--;}
            }
        }
    }

    return count;
}

int brights (float *pix, int n, int k, float thresh) {
  int i=0;
    int blocks = n/k-1;
    int count = 0;
  #pragma acc data create(pix)
    #pragma acc parallel loop reduction(+:count)
    for (i=0;i<blocks;i++){
        count += brightssplit (pix, n, k, thresh, k*i, k*(i+1)+k
    -1);
    }

    count += brightssplit(pix, n, k, thresh, k*i, n);
    return count;
}


int main (int argc, char const *argv[])
{

  float A[MAX_SIZE*MAX_SIZE];
    seed(MAX_SIZE,MAX_SIZE,A);

    int count = 0;
    int i=0;
    for (i=0;i<N_ITERS;i++){
        count += brights (A, 1000, 5, 0.2);
    }

    printf("There are average %d starts in %d iters.\n", count/
    N_ITERS, N_ITERS);
  return 0;
}
```

Listing C.10: Brightcounter_acc

Bibliography

[1] Norm Matloff. *Programming on Parallel Machines.* University of California, Davis, 1993.

[2] Jeff Larkin (2015) *COMPARING OPENACC AND OPENMP PERFORMANCE AND PROGRAMMABILITY.* http://on-demand.gputechconf.com/gtc/2015/presentation/S5196-Jeff-Larkin.pdf

[3] Omni Compiler, *http://omni-compiler.org/*

[4] OpenACC official site, *http://www.openacc.org/*

[5] OpenACC at Nvidia: *https://developer.nvidia.com/openacc*