An Introduction to OpenACC

Zoran Dabic, Rusell Lutrell, Edik Simonian, Ronil Singh, Shrey Tandel

Chapter 1

Introduction

OpenACC is a software accelerator that uses the host and the device. It uses compiler directives in order to give the compiler information about how to parallelize the code. OpenACC divides tasks among gangs. Gangs have workers, and workers have vectors. When the accelerating device is a GPU, a good analogy would be that gangs are blocks, and that workers are warps, and threads are vectors.

The compiler will automatically divide work up in to blocks that have been marked with the kernels or parallel directive, but the programmer can include clauses to be more specific about how data is divided.

Chapter 2

The OpenACC API

2.1 Using OpenACC with C/C++

To begin utilizing OpenACC the user must include the openacc.h header file. GCC 6.1 includes support for openACC. Instructions on how to compile code utilizing openACC are given in appendices A.1 and **??**

2.2 Directives

2.2.1 Grammar

All openACC directives start with **#pragma acc** followed by the directive name and an optional list of clauses. Most directives are followed by a structured block.

In order to motivate descriptions of the following pragmas consider this example of matrix multiplication. This assumes that the caller allocated all the memory for a,b, and r.

```
1
2  void matMult(float * restrict a, float * restrict b, float * restrict r,
       int aRow, int aCol,int bRow, int bCol)
3  {
4     if(aCol != bRow)
5        return;
6     double start;
7     start = omp_get_wtime();
8
9     #pragma acc data copyin(a[0:aCol * aRow],b[0:bCol * bRow]),
          copyout(r[0:aCol * bRow]){
10       #pragma acc parallel{
11          #pragma acc loop
12          for(int row = 0; row < aRow; row++){
13             #pragma acc loop
```

```
14            for(int col = 0; col < bCol; col++){
15
16               float sum = 0;
17               #pragma acc loop reduction(+:sum)
18               for(int i = 0; i < aCol; i++){
19                  sum += a[row * aCol + i] * b[i * bCol + col];
20               }
21               r[row * bCol + col] = sum;
22            }
23         }
24      }
25   }
26   printf("%lf\n",omp_get_wtime() - start );
27 }
```

2.2.2 #pragma acc parallel

Tells the compiler to generate parallelize the code block.Gangs of workers are created to run the code in the block. Code not in a loop is run in gang-redundant mode which executes the same code across all gangs. Parallelism is achieved in loops by splitting the work among several gangs which further split work among their workers.

Consider line 10 which creates a parallel block for the for loop. The work in the for loop is split among a number of gangs. Since the gang number was not specified, gang number will be left to the implementation. The number of gangs and number of workers can be specified by the num_gangs() and num_workers() clauses respectively.

copyin(var-list)

Example in line 9:

This important clause tells the compiler to insert code that copies the data referenced by the variables in the var-list from the system memory to the device memory. The size of the arrays copied should be specified, but for dynamically allocated arrays a pointer is also fine. One can also specify a part of the array to copy as well. The data must be stored contiguously.

copyout(var-list)

The compiler will place code that copies data from the device to system memory after the parallel block is finished executing.

create(var-list)

This clause simply allocates memory on the device. The memory is deallocated on exit from the bock. If an enter data directive was used with the create clause then the delete clause with the exit data directive must be used for deallocation.

Other Data Clauses

There is a series of clauses such as present, present_or_copy, etc that first to check to see if data referenced in the var-list is already present on the device. If so the existing data may be used such as with present or the existing data be be overwritten such as with present_or_copy.

2.2.3 #pragma acc kernels

In contrast to parallel, this pragma has the compiler attempt to infer if a region is parallelizable. The **restrict** keyword should be used in the declaration of all pointers that are to be used in a kernels code block. This keyword tells the compiler that data pointed to by the pointer will only be accessed through the pointer for its lifetime. In short, the **restrict** keyword provides information to help infer potentially parallel code.

2.2.4 #pragma acc loop

This pragma explicitly declares a loop to be parallelizable and allows the programmer to define the type of parallization among other parameters. Loop also allows one to explicitly declare nested loop parallelism. Most compilers support up to 3 levels of nesting.

reduction(<operator>:<scalar var>)

The innermost loop on line 18 would not be parallelizable without this clause. The reduction operator copies a variable into each thread that processes a portion of the loop. After the loop is done executing, every thread then combines their local copies of the variable with the operator specified in the reduction. In matmult's case the total sum is computed.

collapse

In lieu of explicitly marking nested loops with the loop pragma, the outermost loop pragma can specify how many nested loops it applies to using collapse.

independent

If using kernels, this clause tells the compiler that there are no data dependencies between iterations in a loop.

auto

If using kernels, the auto clause tells the compiler to choose what kind of parallelization(group,worker,vector) to use for a loop. The loop must also have the independent clause for actual parallelization to be achieved.

2.2.5 Matrix Multiplication Performance

N for an N * N matrix	GPU parallel	CPU serial
100	0.000579	0.000688
200	0.001985	0.005203
300	0.005215	0.018256
400	0.012431	0.045270
500	0.023382	0.086037
600	0.040141	0.153785
700	0.058165	0.245943
800	0.083303	0.383330
900	0.103303	0.527013
1000	0.239073	0.755712

The speedup is there but could potentially be improved upon.

Chapter 3

Examples

3.1 Example 1: NMF Decomposition

3.1.1 Files:

- nmf.c
- Makefile

3.1.2 What is NMF?

NMF is non-negative matrix factorization. This is an efficient way of storing a matrix by storing it in two smaller sub matrices. Multiplying these two smaller sub matrices gives an approximation of the original matrix. This can be used for compression of data and image files.

3.1.3 Parallelization of NMF

In this example we provided OpenACC directives which optimized the code at a high level to run on the CPU or GPU in parallel. We used the *routine*, *parallel*, and *copy* directives which allowed us to parallize our already existing nmf code.

The *routine* directive was used before the declaration of a function and it tells the compiler to send the function over to the device which will be using this function, in the case of the CPU this function already exists in CPU memory so it is not needed but in the case of the GPU the function must be copied over.

The *gang* clause added at the end of the *routine* directive tells the compiler that the functions contain code that can be parallized. The *seq* clause would be used if the function could not be parallized.

3.1.4 Compiling using pgcc

The nmf code was compiled using pgcc in a PGI workstation. The nmf code was originally in C++ and it was converted to C because the windows PGI workstation did not support pgc++. Compilation time is when the target device is specified. Compiling with *-ta=nvidia* tells the code to execute the parallel portion on the GPU while *-ta=host* tells the code to execute on the CPU in parallel. Additionally *-Minfo* gives information on how the directives were handled. The compiler gives information on what happened at each directive call with the *-Minfo*.

3.1.5 Challenges with OpenACC

The C code for nmf used a struct to store matrices. This is perfectly fine for the CPU but due to OpenACC trying to handle everything at a high level, copying over a user defined type to the GPU results in bad pointers and there is no OpenACC directive to allow for a deep copy of the struct. This means that any dynamically allocated memory in the struct will not be copied over, a user defined function must be made to copy over the data.

Alternatively the data can be copied over by creating a separate variable outside the struct class and making that variable point to the variable in the struct. The latter was used in nmf.c to properly move data over to the GPU memory. Beyond this, if your struct contains variables of other structs then it is advised to change the code so it doesn't. This is because keeping track of data and copying over the data become difficult to keep track of.

3.2 Example 2: Image Recognition

3.2.1 Files:

- /example2/Male/*.txt
- /example2/Female/*.txt
- /example2/Test/*.txt
- /example2/Makefile
- /example2/acc.c
- /example2/bench.sh

3.2.2 Compiler and Environment's settings

For this examples we used OpenACC toolkit provided by Nvidia. This package is from the Portland Group and contains pgcc. Pgcc provides acceleration directives to target cpu/gpu.

3.2.3 Makefile

The Makefile is written in a way to produce three different executables. First, acc.out file is generated without any -acc command, meaning its simple serial version. Second output is acc_cpu.out, which is cpu accelerated using -ta=multicore. Third output is acc_gpu.out, which is gpu accelerated using -ta=nvidia. There are more acceleration target but for this example we just have these three.

3.2.4 Java to C conversion

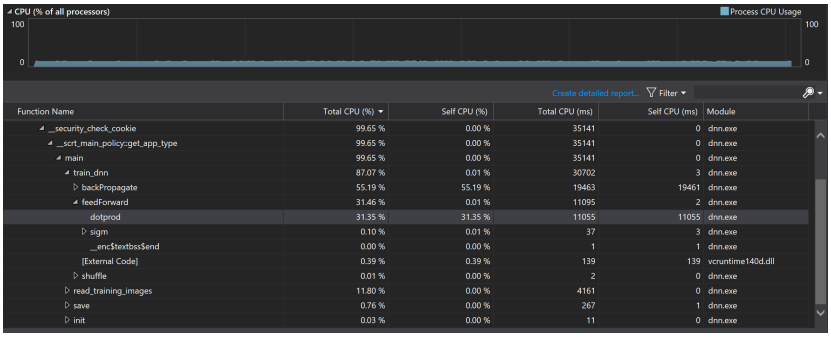
The conversation to C of the existing code was a bit challenging. In order to have the program working in the serial version, we chose to use dynamic allocated arrays of floats. Since these arrays were multi dimensional and each dimension could be different from another, we used pointers to define our multidimensional arrays.

3.2.5 CPU performance profiling

Once the serial version of the program was ready, we used Microsoft Studio performance profiling tool to determine which part of the code is using cpu. We found out that the out dotprod function was consuming %30 of cpu!

3.2.6 CPU targeted parallelization

The most direct way to parallelize the dotprod function was to use #pragma acc kernels loop.



3.2.7 GPU targeted parallelization

For the gpu parallelization we experienced problems. Since our array is multidimensional the copy directive for openacc didn't work as it should. This problem can be solved with two different approaches.

First by creating a deep copy function that takes the multidimensional array and copies it to the host.

Second by changing the data structure of our multidimensional arrays and make them flat contiguous regions of memory so openacc could copy them into the host.

We went with the second solution. However due to complications we couldn't get our code to work as far as gpu parallelization.

3.2.8 Benchmarking process

For the benchmarking we created a custom script called bench.sh. This file runs each variation of the executable output for 10 times and reports the best time back to the user. Followings are the output ran on a laptop that had Nvidia 750M GPU.

Testing serial code...
1 loops, best of 10: 14.8 sec per loop
Testing cpu code...
1 loops, best of 10: 10.3 sec per loop
Testing gpu code...
1 loops, best of 10: 35.5 sec per loop

By the results we can see 4.5 seconds or 35.85% improvement in overall runtime of our program. This is a very good achievement in cpu targeted parallelization because some of the time spent is I/O for reading the images.

3.2.9 Conclusion

In conclusion we see 35% improvement in our program using OpenACC. Even though this is just for cpu version, we can predict around 10x improvement for the gpu version.

3.3 Example 3: SAXPY and Reductions

This is a simple program demonstrating the loop construct and the reduction clause of OpenACC. Here, the `saxpy()` function takes a scalar `a`, two vectors `x` and `y` and applies the function: $y \leftarrow a \cdot x + y$. The next function function, `sum()`, takes a vector `x` and returns the sum of all the values.

In a serial operation, `saxpy()` employs a `for` loop that iterates over each element `i` of `y` and applying `y[i] = a*x[i] + y[i]`. This operation is embarrassingly parallel, so placing `#pragma acc parallel loop` construct above the for loop allows the compiler to spread each iteration across all the available threads. In a similar fashion, the `sum()` employs a for loop to add each element of `x` to a result variable. Since each iteration of the loop is writing to the same result variable, the `reduce` clause of the `loop` directive is used. This instructs the compiler to create multiple copies of the result variable (as many copies as

there are threads) so that each thread can work on its own chunk of the array. Then at the end each of the local copies are added together then returned.

Appendices

Appendix A

A.1 Compiling openACC with OMNI

Compile by running: ompcc -acc <filename>.

A.2 PGI Compiler

For the -ta=host part, the ta stands for target architecture. Host means you are running the program on the cpu. To compile the program for an nvidia gpu, use -ta=nvidia.

The -Minfo flag provides information about how the compiler divides up work. Here is an example:

```
transpose:
     55, Generating Tesla code
         58, #pragma acc loop gang, vector, worker /* blockIdx.x threadIdx. */
         59, #pragma acc loop seq
     55, Generating acc routine gang
     58, Loop is parallelizable
     59, Loop carried dependence of dr-> prevents parallelization
         Complex loop carried dependence of dm-> prevents parallelization
         Loop carried backward dependence of dr-> prevents vectorization
```

The compiler creates a kernel for each parallel region or kernels region. The compiler gives some indication of how memory is mapped to the device.

```
         58, #pragma acc loop gang, vector, worker /* blockIdx.x threadIdx. */
```

A gang maps to a block. A vector maps to a thread. The compiler cannot parallelize the loop at the beginning of line 59, so that directive has no effect.

You can get more useful information about how the parallelism is mapped to the device by setting the PGI_ACC_TIME environment variable to 1. Here is the output for the timing data:

```
Accelerator Kernel Timing data
C:\Users\shrey\Desktop\ECS158\group\stuff\nmf3.c
  nmfompParallel  NVIDIA  devicenum=0
    time(us): 251,567
    235: compute region reached 250 times
        235: kernel launched 250 times
            grid: [50]  block: [32]
            elapsed time(us): total=136,889,000 max=1,253,000 min=250,000 avg=547,556
    235: data region reached 500 times
        235: data copyin transfers: 2500
             device time(us): total=122,023 max=475 min=4 avg=48
        248: data copyout transfers: 2500
             device time(us): total=129,544 max=633 min=4 avg=51
```

The output also contains more detail on how the parallelism is mapped. The output states that there are 50 blocks to the grid and 32 threads to a block. The output also states the time elapsed inside the GPU.

A.3 Who Did What

Zoran: Created chapter 2 and the matmult code.

Edik: Chapter 3 Example 2.

Shrey: Chapter 3 Example 1.

Ronil: Chapter 3 Example 3.

Russell: Intro and Appendix.