OpenACC

Arthur Lei, Michelle Munteanu, Michael Papadopoulos, Philip Smith

Introduction

For this introduction, we are assuming you are familiar with libraries that use a pragma directive based structure, such as OpenMP, as well as libraries that provide wide, vectorized execution on a device with its own memory space, as provided by CUDA. While OpenACC also supports Fortran, we will be using C as our sole programming language.

Open Accelerator, or OpenACC, allows you to focus on writing parallel code, knowing that it will run efficiently on both CPUs and Nvidia GPUs. This library allows for intermixing of device and host code and greatly simplifies the data transfer process.

Like OpenMP, OpenACC uses pragma directives to transform the code on compilation. Unlike OpenMP, it also has directives for managing and copying data. We will revisit this topic in greater depth later.

OpenACC directives are easy to identify in code, as they always begin with the following:

#pragma acc

Before we begin, its worthwhile to remember the differences between parallel programming for a multicore CPU, as seen with OpenMP, and programming for an external accelerator like a GPU.

The most glaring difference is the lack of shared memory. When coding in CUDA, for example, one must remember to manually perform device memory allocation and host-device memory transfers. While OpenACC does not remove the need for memory copying, it hides these implementation details from the programmer, allowing us to focus on our code.

We will begin our exploration with OpenACCs default, naive copying method, leaving more complicated options for later. While this simple method isnt efficient, it is correct. And as youll see, this makes GPU programming almost trivial compared to CUDA.

But the most important detail to remember is that GPUs are adept at performing massively looped operations. Matrices, for example, are a commonly seen use-case. They work best when every thread performs the same job (just on different data). To wit, much of OpenACC is designed around accelerating these kinds of workloads.

GPUs do not do branch-heavy or synchronization-heavy code well. Algorithms requiring entirely independent threads or heavy data sharing will likely perform worse than they would on a multicore CPU. Using a combination of OpenMP and OpenACC allows us to play to each type of hardwares strengths.

OpenACC Basics

There are only a handful of directives needed to make a fully functional, efficient OpenACC program. Like OpenMP, OpenACC allows for clauses to be chained after the basic directive.

Unlike OpenMP, OpenACC also has powerful data lifetime clauses that manage allocating and copying memory.

#pragma acc data makes use of these clauses by simply introducing a scope. Its common to see a data construct enclose the entire program, ensuring that correct data remains on the accelerator regardless of what other directives are being used inside.

#pragma acc parallel is one of two ways to demark accelerator-run code. It works in the same way as OpenMPs parallel pragma. It creates a pool of workers which all redundantly execute the code within. Like OpenMP, enclosing loop directives will cause all workers to work on the loop collaboratively. Forgetting to do so (or intentionally omitting) will result in each worker executing the loop in its entirety.

Unlike the (following) kernels directive, the entire parallel construct compiles to one kernel, which allows for manual work distribution. For example, if you had two consecutive loops, it may be worthwhile for efficiency reasons to force the compiler to produce one kernel instead of allowing it to form two kernels.

But in general, theres little reason to use parallel. Unlike threads on a CPU, a GPU requires many, many threads to have efficient execution. The easiest way to achieve this is to auto-vectorize a loop using the kernels directive.

#pragma acc kernels is the other marking construct. It simply encloses code that will be accelerated by dividing the enclosed code into possibly multiple kernels. Ideally, this should enclose a number of loops which would run sequentially.

By default, unlike parallel, everything runs once, and loops may be analyzed and marked for parallel execution.

Also take heed with placing non-loop code in this structure. It will be compiled into its own kernel and run on the accelerator; it would probably better to drop down to the CPU and execute this code there.

#pragma acc loop is the real powerhouse of OpenACC. Indicating a loop with this directive means that it will be executed across all cores of the accelerator. It performs the same function then, as OpenMPs for directive.

Well start with a simple example: finding palindromes in the first 10 million digits of pi.

The algorithm is straightforward: we treat every digit as a center and test digits to the left and right to grow the palindrome.

As you can see, we make use of the kernels directive and two loops. We also use a data clause on our kernels pragma to handle copying the pi text to the GPU, and our results out. Well go into more detail about these clauses, but for now, you can assume that they operate as intuitively as they appear.

For a more advanced example, well be counting bright spots in a picture.

This algorithm is deceptively simple. We read our image data, filter it to find bright *pixels*, label each bright pixel with a unique set, and finally merge all adjacent sets.

Algorithmically, well give each pixel a number: -1 if its not bright, and its index if it is. This ensures that all bright pixels start out as their own set, with each set indicated by its number. We then loop across all the pixels and if the pixel is bright, attempt to merge its set with its neighbors. Setting each bright pixel to the max of its set number and its neighbors' performs this set-merge operation.

Once weve iterated so that all sets are merged, we transfer this data back to the CPU where it determines the size of the set, and if its large enough, count it towards our total brightspots.

Notice how we used a data directive to keep our image data and our set data in the GPU memory despite going back to the CPU to determine if we should keep iterating.

Something unique to this example is the use of the independent clause on the loop directive. This lets us tell the compiler that wide execution, where elements may be processed simultaneously and out-of-order, is permissible. Without this directive, the compiler infers for itself whether the loop can be run in this wide, parallel way, or that it needs to run the loop sequentially to preserve correctness. Since we are reading and writing to and from the same array (the most trivial example of aliasing), its important to let the compiler know that were allowing these side effects of parallel processing.

In the event you have a loop that must be executed sequentially, theres also the seq clause to denote that. Generally, the compiler will err on the side of caution, but this will be useful if having a sequential loop in a kernels structure is genuinely what you want to do.

Data Synchronization

If youre familiar with OpenMP or CUDA, you might be wondering about barriers. With OpenACC, there are, quite simply, no manual barriers. All data, parallel, and kernels structures have an implicit barrier at the end. Additionally, any kernels formed out the kernels construct have implicit barriers between them. In particular, one must be aware that loop constructs within parallel constructs do **not** have barriers at the end, which is just further incentive to prefer the kernels directive.

Construct Clauses

Proper use of the optional clauses to the **kernels**, **loop**, and **data** constructs is key to the effective use of OpenACC. This section gives a brief overview of some of the most useful clauses. More detailed descriptions of all clauses are given in the OpenACC 1.0 specifications [1]. Clauses not mentioned in prior sections are given brief treatment here.

Kernels and Data Clauses

The following clauses are all exclusive to the kernels (or the combined **kernels loop**) and data constructs covered in this tutorial, and deals exclusively with moving data between the device and host. Of particular note regards OpenACC's treatment of pointers to dynamically allocated arrays:

you must specify what range of values to copy.:

The convention OpenACC adopts for specifying subarrays to copy is:

ptr[startIndex:numElements].

If *startIndex* is omitted, the beginning of the array is assumed (as if *startIndex = 0*). *numElements* is the total number of items to copy starting with *startIndex*.

copyin(var1, var2 ... varN): Blindly copies a list of variables from the host to the device upon entering the construct region.

copyout(var1 var2 ... varN): Blindly copies a list of variables from device to host upon exiting the construct region.

copy(var1, var2, ..., varN): Behaves as if both **copyin** and **copyout** clauses were invoked on the list of variables.

create(var1, var2, ..., varN): Allocates spaces for the list of host variables on the device on entry to the construct region, but does not copy.

present(var1, var2, ..., varN): Declares a list of variables as already present on the device. If a variable cannot be found on the device, the program halts with an error.

present_or_copy(var1, var2, ..., varN): Declares a list of variables as already present on the device - if any of the variables listed are not found, they are allocated and copied from host to device upon entry to the construct region, and then copied from device to host upon exit. Note - this means that if a variable is already present on the device, it will not be copied back to the host on construct region exit.

present_or_copyin(var1, var2, ..., varN): A variant of **present_or_copy**, but with no

[1] Available from `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`

copying from device to host upon construct region exit.

present_or_copyout(var1, var2, ..., varN): A variant of **present_or_copy**, but only allocation occurs if a variable is not present on the device, and the variable is copied from device to host upon construct region exit.

present_or_create(var1, var2, ..., varN): A variant of **present_or_copy**, but only allocation occurs if a variable is not present on the device.

Loop Clauses

Important clauses unique to the **loop** construct (or the combined **kernels loop** construct) are the following:

private(var1, var2, ..., varN): Creates a copy of each variable for each loop iteration.

reduction(operator : var1, var2, ..., varN): Creates a copy of each *scalar* variable for each loop iteration (as in the private, but restricted to scalar values), and applies a reduction operator at the end of the loop - eg: **reduction(+:z)** would sum each loop's instance of variable **z** into **z**. OpenACC 1.0 only supports (for C/C++) **int**, **float**, **double**, and **complex** as valid scalar variables for the reduction clause, as well as the following operators:

C and C++	
operator	initialization value
+	0
*	1
max	least
min	largest
&	~0
|	0
&&	1
||	0

It should be noted that the variables must be declared prior to the loop construct, and that each private copy is initialized to a value that may not be the user-initialized value. In addition, the user-initialized value will also have the reduction operator applied to it after the loop construct exits. For example, were **reduction(*:y)** called, with **y** initialized to 0, the result will always be zero, despite each private **y** being initialized to 1 for each iteration.

π Palindromes

Relevent timing data:

OpenACC: 0.97s[2]

Serial(stripped of #pragmas): 0.96s[3]

For trivial pursuit: the ten longest palindromes in the first ten million digits of of π are:

95715977951759
9475082805749
7139999999317
4072839382704
4246534356424
450197791054
651547745156
559860068955
431275572134
350052250053

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

int compare(const void* a, const void* b) {
  return -(((int*) a)[1] - ((int*) b)[1]);
}

int main() {

  FILE* file = fopen("pi.txt", "r");

  fseek(file, 0, SEEK_END);
  int length = ftell(file);
  fseek(file, 0, SEEK_SET);

  char* data = (char*) malloc(length);
  fread(data, 1, length, file);

  // result matrix
  int resultsLen = 2 * 2 * length;
  int* results = (int*) malloc(resultsLen * sizeof(int));
  int i = 0;
```

[2]ompcc -acc palindrome.c -o palindrome.out
[3]gcc palindromeSERIAL.c palindromeSERIAL.out; ompcc emitted incorrect code

```
#pragma acc kernels copy(data[0:length], results[0:resultsLen])
{
  // odd length palindrome finding
  #pragma acc loop
  for (i = 0; i < length; i++) {
    int j;
    for (j = 1;; j++) {

      int left = i - j;

      if (left < 0)
        break;

      int right = i + j;

      if (right >= length)
        break;

      if (data[left] != data[right])
        break;
    }
    results[2 * i + 0] = i;
    results[2 * i + 1] = 1 + 2 * (j - 1);
  }

  // even length palindrome finding
  #pragma acc loop
  for (i = 0; i < length; i++) {
    int j;
    for (j = 0;; j++) {

      int left = i - j;

      if (left < 0)
        break;

      int right = i + j + 1;

      if (right >= length)
        break;

      if (data[left] != data[right])
        break;
    }
    results[2 * (i + length) + 0] = i;
    results[2 * (i + length) + 1] = 2 * j;
  }
} //acc kernels region

qsort(results, 2 * length, 2 * sizeof(int), compare);

for (int i = 0; i < 10; i++) {

  char* buf = (char*) malloc(results[2 * i + 1] + 1);
```

```
    memset(buf, 0, results[2 * i + 1] + 1);

    if (results[2 * i + 1] % 2 == 1)
      memcpy(buf, data + results[2 * i] - results[2 * i + 1]/2, results[2
          * i + 1]);
    else
      memcpy(buf, data + results[2 * i] - results[2 * i + 1]/2 + 1,
          results[2 * i + 1]);

    printf("%s\n", buf);

    free(buf);

  }
}
```

Brightspot Blobs

Relevent timing data:

OpenACC: 0.79s[4]

Serial(stripped of #pragmas): 0.95s[5]

The test image used was:

[6]

Our algorithm counted 248 bright spots!

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

[4]ompcc -acc brightspots.c -o brightspots.out

[5]gcc brightspotsSERIAL.c brightspotsSERIAL.out; gcc used to be consistent with previous example's tests

[6]Image provided by Philip Smith, released to the public domain

```c
typedef int bool;
#define true 1
#define false 0

int compare(const void* a, const void* b) {
  return ( *(int*)a - *(int*)b );
}

int main() {

  // File processed by the GIMP to be raw RGB ints
  FILE* file = fopen("firework.data", "r");

  // we also hardcode the dimensions
  int width = 4912;
  int height = 3264;
  int size = width * height;

  fseek(file, 0, SEEK_END);
  int length = ftell(file);
  fseek(file, 0, SEEK_SET);

  if (length != size * 3) {
    printf("illegal size\n");
    exit(1);
  }

  unsigned char* data = (char*) malloc(length);
  fread(data, 1, length, file);

  // let's assume a bright pixel is above RGB(235, 235, 235)
  unsigned char threshold = 235;

  // and a hot spot is at least 16 pixels
  int min_size = 16;

  // result matrix
  int* results = (int*) malloc(size * sizeof(int));

  int hot_count = 0;
  int i;
  // hotpixel finding pass
  for (i = 0; i < size; i++) {
    if (
        data[3 * i] >= threshold
        && data[(3 * i) + 1] >= threshold
        && data[(3 * i) + 2] >= threshold
    ) {
      results[i] = i;
      hot_count++;
    } else {
      results[i] = -1;
    }
  }
```

```
bool keep_looping = true;

int loop_count = 0;

#pragma acc data copy(results[0:size])
{
  while (keep_looping) {

    if (loop_count % 10 == 0)
      printf("%d\n", loop_count);

    keep_looping = false;

    #pragma acc kernels loop independent
    for (i = 0; i < size; i++) {
      if (results[i] != -1) {

        if (i + width < size && results[i + width] > results[i]) {
          results[i] = results[i + width];
          keep_looping = true;
        }

        if (i + 1 < size && results[i + 1] > results[i]) {
          results[i] = results[i + 1];
          keep_looping = true;
        }

      }
    } //acc loop
    loop_count++;
  }//while loop
} //acc data region

qsort(results, size, sizeof(int), compare);

for (i = 0; i < size; i++) {
  if (results[i] != -1)
    break;
}

int count = 0;

int last_set = -1;
int set_size;

for (; i < size; i++) {
  if (results[i] != last_set) {

    if (set_size > min_size)
      count++;

    last_set = results[i];
    set_size = 0;
```

```
    }

    set_size++;
  }

  printf("%d hot spots\n", count);

}
```

Who Did What

Arthur: Assisting with original algorithm development, proofreading document.
Michelle: OpenACC research, document editing, proofreading.
Michael: Typesetting/editing into latex and port of original OpenMP code to OpenACC.
Philip: Developed novel example programs in OpenMP.