OpenACC Programming and Best Practices Guide

June 2015

OpenACC
Directives for Accelerators

© 2015 openacc-standard.org. All Rights Reserved.

Contents

Chapter 1

Introduction

This guide presents methods and best practices for accelerating applications in an incremental, performance portable way. Although some of the examples may show results using a given compiler or accelerator, the information presented in this document is intended to address all architectures both available at publication time and well into the future. Readers should be comfortable with C, C++, or Fortran, but do not need experience with parallel programming or accelerated computing, although such experience will be helpful.

Writing Portable Code

The current computing landscape is spotted with a variety of computing architectures: multi-core CPUs, GPUs, many-core devices, DSPs, and FPGAs, to name a few. It is now commonplace to find not just one, but several of these differing architectures within the same machine. Programmers must make portability of their code a forethought, otherwise they risk locking their application to a single architecture, which may limit the ability to run on future architectures. Although the variety of architectures may seem daunting to the programmer, closer analysis reveals trends that show a lot in common between them. The first thing to note is that all of these architectures are moving in the direction of more parallelism. CPUs are not only adding CPU cores but also expanding the length of their SIMD operations. GPUs have grown to require a high degree of block and SIMT parallelism. It is clear that going forward all architectures will require a significant degree of parallelism in order to achieve high performance. Modern processors need not only a large amount of parallelism, but frequently expose multiple levels of parallelism with varying degrees of coarseness. The next thing to notice is that all of these architectures have exposed hierarchies of memory. CPUs have the main system memory, typically DDR, and multiple layers of cache memory. GPUs have the main CPU memory, the main GPU memory, and various degrees of cache or scratchpad memory. Additionally on hybrid architectures, which include two or more different architectures, there exist machines where the two architectures have completely separate memories, some with physically separate but logically the same memory, and some with fully shared memory.

Because of these complexities, it's important that developers choose a programming model that balances the need for portability with the need for performance. Below are four programming models of varying degrees of both portability and performance. In a real application it's frequently best to use a mixture of approaches to ensure a good balance between high portability and performance.

Libraries

Standard (and defacto standard) libraries provide the highest degree of portability because the programmer can frequently replace only the library used without even changing the source code itself when changing compute architectures. Since many hardware vendors provide highly-tuned versions of common libraries, using libraries can also result in very high performance. Although libraries can provide both high portability and high performance, few applications are able to use only libraries because of their limited scope.

Some vendors provide additional libraries as a value-added for their platform, but which implement non-standard APIs. These libraries provide high performance, but little portability. Fortunately because libraries provide modular APIs, the impact of using non-portable libraries can be isolated to limit the impact on overall application portability.

Standard Programming Languages

Many standard programming languages either have or are beginning to adopt features for parallel programming. For example, Fortran 2008 added support for `do concurrent`, which exposes the parallelism within that loop. Adoption of these language features is often slow, however, and many standard languages are only now beginning to discuss parallel programming features for future language releases. When these features become commonplace, they will provide high portability, since they are part of a standard language, and if well-designed can provide high performance as well.

Compiler Directives

When standard programming languages lack support for necessary features compiler directives can provide additional functionality. Directives, in the form of pragmas in C/C++ and comments in Fortran, provide additional information to compilers on how to build and/or optimize the code. Most compilers support their own directives, but also directives such as OpenACC and OpenMP, which are backed by industry groups and implemented by a range of compilers. When using industry-backed compiler directives the programmer can write code with a high degree of portability across compilers and architectures. Frequently, however, these compiler directives are written to remain very high level, both for simplicity and portability, meaning that performance may lag lower-level programming paradigms. Many developers are willing to give up 10-20% of hand-tuned performance in order to get a high degree of portability to other architectures and to enhance programmer productivity. The tolerance for this portability/performance trade-off will vary according to the needs of the programmer and application.

Parallel Programming Extensions

CUDA and OpenCL are examples of extensions to existing programming languages to give additional parallel programming capabilities. Code written in these languages is frequently at a lower level than that of other options, but as a result can frequently achieve higher performance. Lower level architectural details are exposed and the way that a problem is decomposed to the hardware must be explicitly managed with these languages. This is the best option when performance goals outweigh portability, as the low-level nature of these programming languages frequently makes the resulting code less portable. Good software engineering practices can reduce the impact these languages have on portability.

There is no one programming model that fits all needs. An application developer needs to evaluate the priorities of the project and make decisions accordingly. A best practice is to begin with the most portable and productive programming models and move to lower level programming models only as needed and in a modular fashion. In doing so the programmer can accelerate much of the application very quickly, which is often more beneficial than attempting to get the absolute highest performance out of a particular routine before moving to the next. When development time is limited, focusing on accelerating as much of the application as possible is generally more productive than focusing solely on the top time consuming routine.

What is OpenACC?

With the emergence of GPU and many-core architectures in high performance computing, programmers desire the ability to program using a familiar, high level programming model that provides both high performance and portability to a wide range of computing architectures. OpenACC emerged in 2011 as a programming model that uses high-level compiler directives to expose parallelism in the code and parallelizing compilers to build the code for a variety of parallel accelerators. This document is intended as a best practices guide for accelerating an application using OpenACC to give both good performance and portability to other devices.

The OpenACC Accelerator Model

In order to ensure that OpenACC would be portable to all computing architectures available at the time of its inception and into the future, OpenACC defines an abstract model for accelerated computing. This model exposes multiple levels of parallelism that may appear on a processor as well as a hierarchy of memories with varying degrees of speed and addressibility. The goal of this model is to ensure that OpenACC will be applicable to more than just a particular architecture or even just the architectures in wide availability at the time, but to ensure that OpenACC could be used on future devices as well.

At its core OpenACC supports offloading of both computation and data from a *host* device to an *accelerator* device. In fact, these devices may be the same or may be completely different architectures, such as the case of a CPU host and GPU accelerator. The two devices may also have separate memory spaces or a single memory space. In the case that the two devices have different memories the OpenACC compiler and runtime will analyze the code and handle any accelerator memory management and the transfer of data between host and device memory. Figure 1.1 shows a high level diagram of the OpenACC abstract accelerator, but remember that the devices and memories may be physically the same on some architectures.

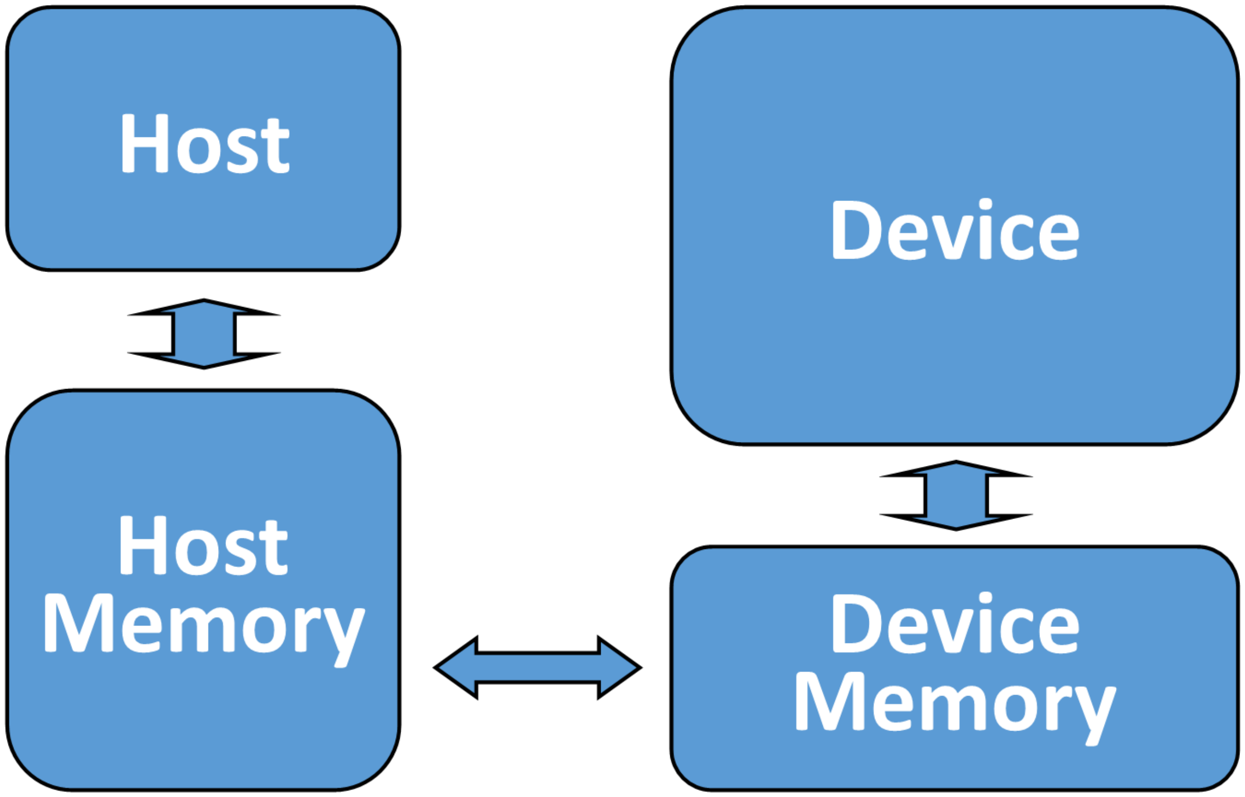


Figure 1.1: OpenACC's Abstract Accelerator Model

More details of OpenACC's abstract accelerator model will be presented throughout this guide when they are pertinent.

Best Practice: For developers coming to OpenACC from other accelerator programming models, such as CUDA or OpenCL, where host and accelerator memory is frequently represented by two distinct variables (`host_A[]` and `device_A[]`, for instance), it's important to remember that when using OpenACC a variable should be thought of as a single object, regardless of whether the it's backed by memory in one or more memory spaces. If one assumes that a variable represents two separate memories, depending on where it is used in the program, then it is possible to write programs that access the variable in unsafe ways, resulting in code that would not be portable to devices that share a single memory between the host and device. As with any parallel or asynchronous programming paradigm, accessing the same variable from two sections of code simultaneously could result in a race condition that produces inconsistent results. By assuming that you are always accessing a single variable, regardless of how it is stored in memory, the programmer will avoid making mistakes that could cost a significant amount of effort to debug.

Benefits and Limitations of OpenACC

OpenACC is designed to be a high-level, platform independent language for programming accelerators. As such, one can develop a single source code that can be run on a range of devices and achieve good performance. The simplicity and portability that OpenACC's programming model provides sometimes comes at a cost to performance. The OpenACC abstract accelerator model defines a least common denominator for accelerator devices, but cannot represent architectural specifics of these devices without making the language less portable. There will always be some optimizations that are possible in a lower-level programming model, such as CUDA or OpenCL, that cannot be represented at a high level. For instance, although OpenACC has the `cache` directive, some uses of *shared memory* on NVIDIA GPUs are more easily represented using CUDA. The same is true for any host or device: certain optimizations are too low-level for a high-level approach like OpenACC. It is up to the developers to determine the cost and benefit of selectively using a lower level programming language for performance critical sections of code. In cases where performance is too critical to take a high-level approach, it's still possible to use OpenACC for much of the application, while using another approach in certain places, as will be discussed in a later chapter on interoperability.

Accelerating an Application with OpenACC

This section will detail an incremental approach to accelerating an application using OpenACC. When taking this approach it is beneficial to revisit each step multiple times, checking the results of each step for correctness. Working incrementally will limit the scope of each change for improved productivity and debugging.

OpenACC Directive Syntax

This guide will introduce OpenACC directives incrementally, as they become useful for the porting process. All OpenACC directives have a common syntax, however, with the `acc` sentinal, designating to the compiler that the text that follows will be OpenACC, a directive, and clauses to that directive, many of which are optional but provide the compiler with additional information.

In C and C++, these directives take the form of a pragma. The example code below shows the OpenACC `kernels` directive without any additional clauses

```
#pragma acc kernels
```

In Fortran, the directives take the form of a special comment, as demonstrated below.

```
    !$acc kernels
```

Some OpenACC directives apply to structured blocks of code, while others are executable statements. In C and C++ a block of code can be represented by curly braces (`{` and `}`). In Fortran a block of code will begin with an OpenACC directive (`!$acc kernels`) and end with a matching ending directive (`!$acc end kernels`).

Porting Cycle

Programmers should take an incremental approach to accelerating applications using OpenACC to ensure correctness. This guide will follow the approach of first assessing application performance, then using OpenACC to parallelize important loops in the code, next optimizing data locality to remove unnecessary data migrations between the host and accelerator, and finally optimizing loops within the code to maximize performance on a given architecture. This approach has been successful in many applications because it prioritizes changes that are likely to provide the greatest returns so that the programmer can quickly and productively achieve the acceleration.

There are two important things to note before detailing each step. First, at times during this process application performance may actually slow down. Developers should not become frustrated if their initial efforts result in a loss of performance. As will be explained later, this is generally the result of implicit data movement between the host and accelerator, which will be optimized as a part of the porting cycle. Second, it is critical that developers check the program results for correctness after each change. Frequent correctness checks will save a lot of debugging effort, since errors can be found and fixed immediately, before they have the chance to compound. Some developers may find it beneficial to use a source version control tool to snapshot the code after each successful change so that any breaking changes can be quickly thrown away and the code returned to a known good state.

Assess Application Performance

Before one can begin to accelerate an application it is important to understand in which routines and loops an application is spending the bulk of its time and why. It is critical to understand the most time-consuming parts of the application to maximize the benefit of acceleration. Amdahl's Law informs us that the speed-up achievable from running an application on a parallel accelerator will be limited by the remaining serial code. In other words, the application will see the most benefit by accelerating as much of the code as possible and by prioritizing the most time-consuming parts. A variety of tools may be used to identify important parts of the code, including simple application timers.

Parallelize Loops

Once important regions of the code have been identified, OpenACC directives should be used to accelerate these regions on the target device. Parallel loops within the code should be decorated with OpenACC directives to provide OpenACC compilers the information necessary to parallelize the code for the target architecture.

Optimize Data Locality

Because many accelerated architectures, such as CPU + GPU architectures, use distinct memory spaces for the *host* and *device* it is necessary for the compiler to manage data in both memories and move the data between the two memories to ensure correct results. Compilers rarely have full knowledge of the application, so they must be cautious in order to ensure correctness, which often involves copying data to and from the accelerator more often than is actually necessary. The programmer can give the compiler additional information about how to manage the memory so that it remains local to the accelerator as long as possible and is only moved between the two memories when absolutely necessary. Programmers will often realize the largest performance gains after optimizing data movement during this step.

Optimize Loops

Compilers will make decisions about how to map the parallelism in the code to the target accelerator based on internal heuristics and the limited knowledge it has about the application. Sometimes additional performance can be gained by providing the compiler with more information so that it can make better decisions on how to map the parallelism to the accelerator. When coming from a traditional CPU architecture to a more parallel architecture, such as a GPU, it may also be necessary to restructure loops to expose additional parallelism for the accelerator or to reduce the frequency of data movement. Frequently code refactoring that was motivated by improving performance on parallel accelerators is beneficial to traditional CPUs as well.

This process is by no means the only way to accelerate using OpenACC, but it has been proven successful in numerous applications. Doing the same steps in different orders may cause both frustration and difficulty debugging, so it's advisable to perform each step of the process in the order shown above.

Heterogenous Computing Best Practices

Many applications have been written with little or even no parallelism exposed in the code. The applications that do expose parallelism frequently do so in a coarse-grained manner, where a small number of threads or processes execute for a long time and compute a significant amount work each. Modern GPUs and many-core processors, however, are designed to execute fine-grained threads, which are short-lived and execute a minimal amount of work each. These parallel architectures achieve high throughput by trading single-threaded performance in favor of several orders in magnitude more parallelism. This means that when accelerating an application with OpenACC, which was designed in light of increased hardware parallelism, it may be necessary to refactor the code to favor tightly-nested loops with a significant amount of data reuse. In many cases these same code changes also benefit more traditional CPU architectures as well by improving cache use and vectorization.

OpenACC may be used to accelerate applications on devices that have a discrete memory or that have a memory space that's shared with the host. Even on devices that utilize a shared memory there is frequently still a hierarchy of a fast, close memory for the accelerator and a larger, slower memory used by the host. For this reason it is important to structure the application code to maximize reuse of arrays regardless of whether the underlying architecture uses discrete or unified memories. When refactoring the code for use with OpenACC it is frequently beneficial to assume a discrete memory, even if the device you are developing on has a unified memory. This forces data locality to be a primary consideration in the refactoring and will ensure that the resulting code exploits hierarchical memories and is portable to a wide range of devices.

Case Study - Jacobi Iteration

Throughout this guide we will use simple applications to demonstrate each step of the acceleration process. The first such application will solve the 2D-Laplace equation with the iterative Jacobi solver. Iterative methods are a common technique to approximate the solution of elliptic PDEs, like the 2D-Laplace equation, within some allowable tolerance. In the case of our example we will perform a simple stencil calculation where each point calculates it value as the mean of its neighbors' values. The calculation will continue to iterate until either the maximum change in value between two iterations drops below some tolerance level or a maximum number of iterations is reached. For the sake of consistent comparison through the document the examples will always iterate 1000 times. The main iteration loop for both C/C++ and Fortran appears below.

```c
    while ( error > tol && iter < iter_max )
    {
        error = 0.0;

        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

        iter++;
    }
```

```fortran
    do while ( error .gt. tol .and. iter .lt. iter_max )
      error=0.0_fp_kind

      do j=1,m-2
        do i=1,n-2
          Anew(i,j) = 0.25_fp_kind * ( A(i+1,j  ) + A(i-1,j  ) + &
                                       A(i  ,j-1) + A(i  ,j+1) )
          error = max( error, abs(Anew(i,j)-A(i,j)) )
        end do
      end do

      do j=1,m-2
        do i=1,n-2
          A(i,j) = Anew(i,j)
        end do
      end do

      if(mod(iter,100).eq.0 ) write(*,'(i5,f10.6)'), iter, error
      iter = iter + 1

    end do
```

The outermost loop in each example will be referred to as the *convergence loop*, since it loops until the answer has converged by reaching some maximum error tolerance or number of iterations. Notice that whether or not a loop iteration occurs depends on the error value of the previous iteration. Also, the values for each element of `A` is calculated based on the values of the previous iteration, known as a data dependency. These two facts mean that this loop cannot be run in parallel.

The first loop nest within the convergence loop calculates the new value for each element based on the current values of its neighbors. Notice that it is necessary to store this new value into a different array. If each iteration stored the new value back into itself then a data dependency would exist between the data elements, as the order each element is calculated would affect the final answer. By storing into a temporary array we ensure that all values are calculated using the current state of `A` before `A` is updated. As a result, each loop iteration is completely independent of each other iteration. These loop iterations may safely be run in any order or in parallel and the final result would be the same. This loop also calculates a maximum error value. The error value is the difference between the new value and the old. If the maximum amount of change between two iterations is within some tolerance, the problem is considered converged and the outer loop will exit.

The second loop nest simply updates the value of `A` with the values calculated into `Anew`. If this is the last iteration of the convergence loop, `A` will be the final, converged value. If the problem has not yet converged, then `A` will serve as the input for the next iteration. As with the above loop nest, each iteration of this loop nest is independent of each other and is safe to parallelize.

In the coming sections we will accelerate this simple application using the method described in this document.

Chapter 2

Assess Application Performance

A variety of tools can be used to evaluate application performance and which are available will depend on your development environment. From simple application timers to graphical performance analyzers, the choice of performance analysis tool is outside of the scope of this document. The purpose of this section is to provide guidance on choosing important sections of code for acceleration, which is independent of the profiling tools available.

Because this document is focused on OpenACC, the PGProf tool, which is provided with the PGI OpenACC compiler will be used for CPU profiling. When accelerator profiling is needed, the application will be run on an Nvidia GPU and the Nvidia Visual Profiler will be used.

Baseline Profiling

Before parallelizing an application with OpenACC the programmer must first understand where time is currently being spent in the code. Routines and loops that take up a significant percentage of the runtime are frequently referred to as *hot spots* and will be the starting point for accelerating the application. A variety of tools exist for generating application profiles, such as gprof, pgprof, Vampir, and TAU. Selecting the specific tool that works best for a given application is outside of the scope of this document, but regardless of which tool or tools are used below are some important pieces of information that will help guide the next steps in parallelizing the application.

- Application performance - How much time does the application take to run? How efficiently does the program use the computing resources?
- Program hotspots - In which routines is the program spending most of its time? What is being done within these important routines? Focusing on the most time consuming parts of the application will yield the greatest results.
- Performance limiters - Within the identified hotspots, what's currently limiting the application performance? Some common limiters may be I/O, memory bandwidth, cache reuse, floating point performance, communication, etc. One way to evaluate the performance limiters of a given loop nest is to evaluate its *computational intensity*, which is a measure of how many operations are performed on a data element per load or store from memory.
- Available parallelism - Examine the loops within the hotspots to understand how much work each loop nest performs. Do the loops iterate 10's, 100's, 1000's of times (or more)? Do the loop iterations operate independently of each other? Look not only at the individual loops, but look a nest of loops to understand the bigger picture of the entire nest.

Gathering baseline data like the above both helps inform the developer where to focus efforts for the best results and provides a basis for comparing performance throughout the rest of the process. It's important to choose input that will realistically reflect how the application will be used once it has been accelerated. It's tempting to use a known benchmark problem for profiling, but frequently these benchmark problems use a reduced problem size or reduced I/O, which may lead to incorrect assumptions about program performance. Many developers also use the baseline profile to gather the expected output of the application to use for verifying the correctness of the application as it is accelerated.

Additional Profiling

Through the process of porting and optimizing an application with OpenACC it's necessary to gather additional profile data to guide the next steps in the process. Some profiling tools, such as pgprof and Vampir, support profiling on CPUs and GPUs, while other tools, such as gprof and NVIDIA Visual Profiler, may only support profiling on a particular platform. Additionally, some compilers build their own profiling into the application, such is the case with the PGI compiler, which supports setting the PGI_ACC_TIME environment variable for gathering runtime information about the application. When developing on offloading platforms, such as CPU + GPU platforms, it's generally important to use a profiling tool throughout the development process that can evaluate both time spent in computation and time spent performing PCIe data transfers. This document will use NVIDIA Visual Profiler for performing this analysis, although it is only available on NVIDIA platforms.

Case Study - Analysis

To get a better understanding of the case study program we will use the PGProf utility that comes as a part of the PGI Workstation package. First, it's necessary to build the executable to embed the compiler feedback into the executable using the *common compiler feedback framework* (CCFF) feature of the PGI compiler. This feature is enabled with the `-Mprof=ccff` compiler flag and embeds additional information into the executable that can then be used by the PGProf utility to display additional information about how the compiler optimized the code. The executable is built with the following command:

```
$ pgcc -fast -Minfo=all -Mprof=ccff laplace2d.c
main:
     41, Loop not fused: function call before adjacent loop
         Loop not vectorized: may not be beneficial
         Unrolled inner loop 4 times
         Generated 3 prefetches in scalar loop
     58, Generated an alternate version of the loop
         Generated vector sse code for the loop
         Generated 3 prefetch instructions for the loop
     68, Memory copy idiom, loop replaced by call to __c_mcopy8
```

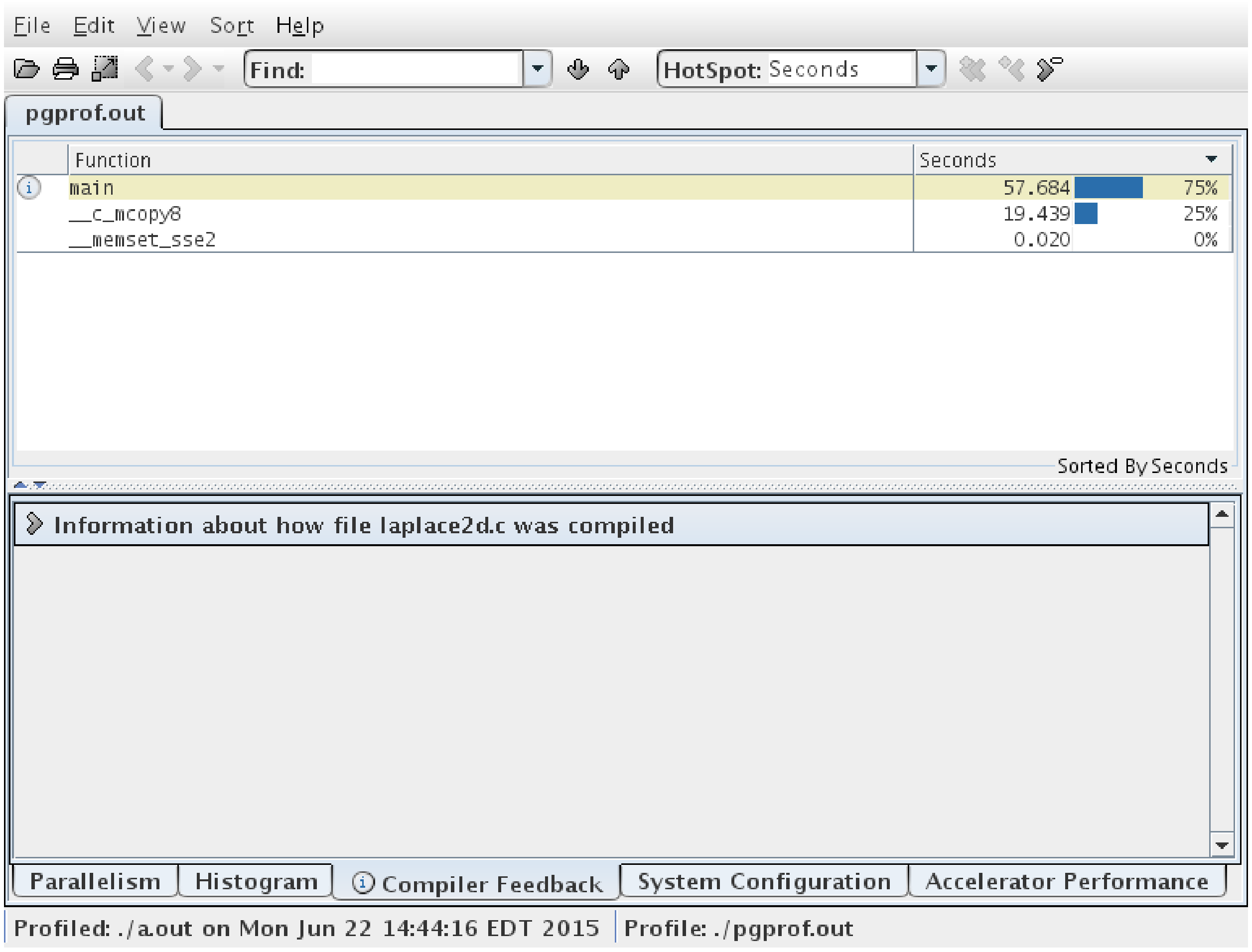
Once the executable has been built, the `pgcollect` command will run the executable and gather information that can be used by PGProf to profile the executable.

```
$ pgcollect ./a.out
Jacobi relaxation Calculation: 4096 x 4096 mesh
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000403
  700, 0.000345
  800, 0.000302
  900, 0.000269
 total: 76.340051 s
target process has terminated, writing profile data
```

Once the data has been collected, it can be visualized using the `pgprof` command, which will open a PGProf window.

```
$ pgprof -exe ./a.out
```

When PGPROG opens we see that the vast majority of the time is spent in two routines: main and ___c_mcopy8. A screenshot of the initial screen for PGProf is show in figure 2.1. Since the code for this case study is completely within the main function of the program, it's not surprising that nearly all of the time is spent in main, but in larger applications it's likely that the time will be spent in several other routines.



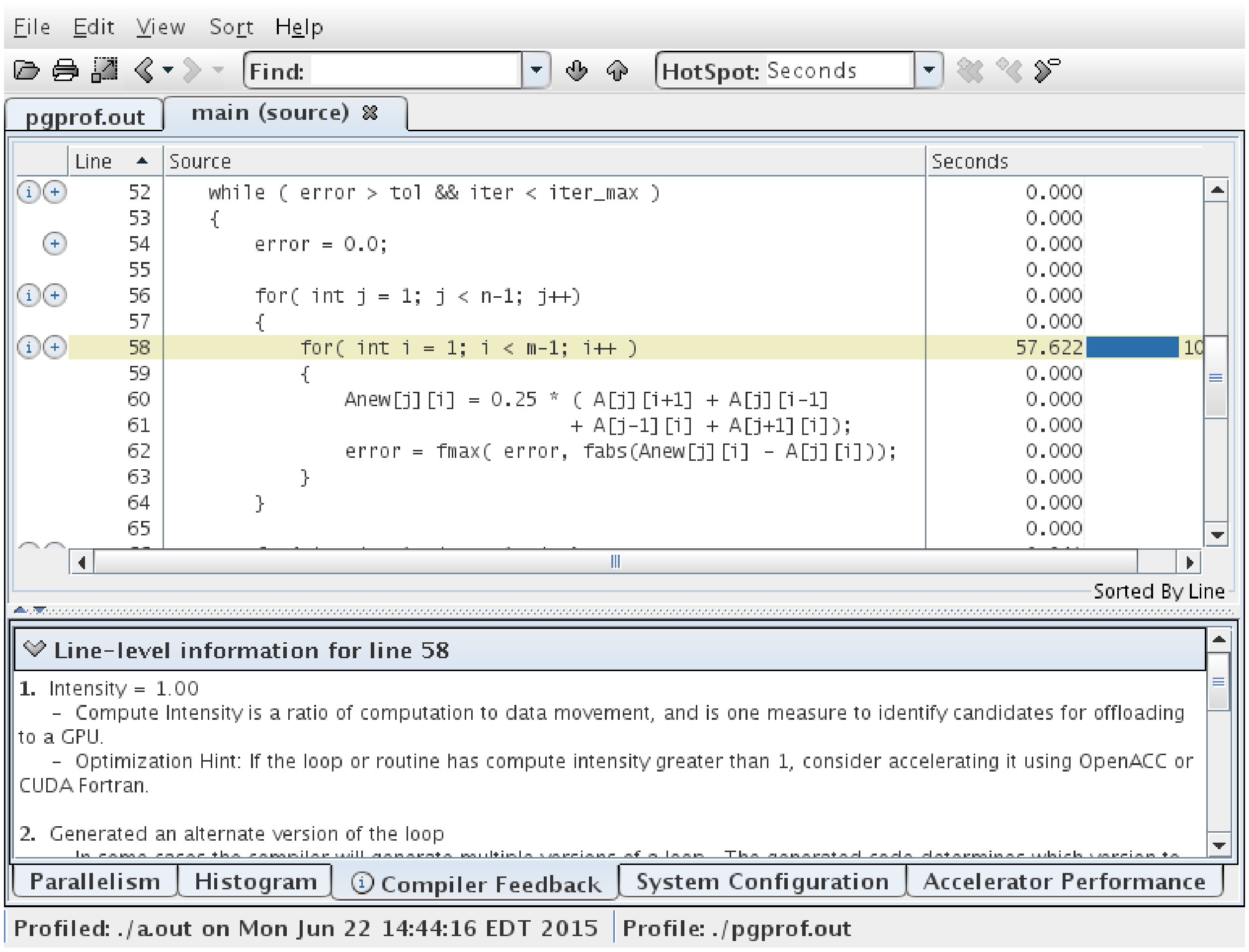
Function	Seconds	
main	57.684	75%
__c_mcopy8	19.439	25%
__memset_sse2	0.020	0%

Information about how file laplace2d.c was compiled

Profiled: ./a.out on Mon Jun 22 14:44:16 EDT 2015 | Profile: ./pgprof.out

Figure 2.1: PGProf initial profile window showing 75% of runtime in main and 25% in a memory copy routine.

Clicking into the main function we can see that nearly all of the runtime within main comes from the loop that calculates the next value for A. This is shown in figure 2.2. What is not obvious from the profiler output, however, is that the time spent in the memory copy routine shown in the initial screen is actually the second loop nest, which performs the array swap at the end of each iteration. The compiler output shows above (and is reiterated in PGProf) that the loop at line 68 was replaced by a memory copy, because doing so is more efficient than copying each element individually. So what the profiler is really showing us is that the major hotspots for our application are the loop nest that calculate `Anew` from `A` and the loop nest that copies from `Anew` to `A` for the next iteration, so we'll concentrate our efforts on these two loop nests.



```
Line  Source                                                        Seconds
 52     while ( error > tol && iter < iter_max )                    0.000
 53     {                                                           0.000
 54         error = 0.0;                                            0.000
 55                                                                 0.000
 56         for( int j = 1; j < n-1; j++)                           0.000
 57         {                                                       0.000
 58             for( int i = 1; i < m-1; i++ )                      57.622  10
 59             {                                                   0.000
 60                 Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]     0.000
 61                                     + A[j-1][i] + A[j+1][i]);   0.000
 62                 error = fmax( error, fabs(Anew[j][i] - A[j][i])); 0.000
 63             }                                                   0.000
 64         }                                                       0.000
 65                                                                 0.000
```

Sorted By Line

Line-level information for line 58

1. Intensity = 1.00
– Compute Intensity is a ratio of computation to data movement, and is one measure to identify candidates for offloading to a GPU.
– Optimization Hint: If the loop or routine has compute intensity greater than 1, consider accelerating it using OpenACC or CUDA Fortran.

2. Generated an alternate version of the loop

Profiled: ./a.out on Mon Jun 22 14:44:16 EDT 2015 | Profile: ./pgprof.out

Figure 2.2: PGProf analysis of the major compute kernel within the main function

In the chapters that follow, we will optimize the loops identified in this chapter as the hotspots within our example application.

Chapter 3

Parallelize Loops

Now that the important hotspots in the application have been identified, the programmer should incrementally accelerate these hotspots by adding OpenACC directives to the important loops within those routines. There is no reason to think about the movement of data at this point in the process, the OpenACC compiler will analyze the data needed in the identified region and automatically ensure that the data is available on the accelerator. By focusing solely on the parallelism during this step, the programmer can move as much computation to the device as possible and ensure that the program is still giving correct results before optimizing away data motion in the next step. During this step in the process it is common for the overall runtime of the application to increase, even if the execution of the individual loops is faster using the accelerator. This is because the compiler must take a cautious approach to data movement, frequently copying more data to and from the accelerator than is actually necessary. Even if overall execution time increases during this step, the developer should focus on expressing a significant amount of parallelism in the code before moving on to the next step and realizing a benefit from the directives.

OpenACC provides two different approaches for exposing parallelism in the code: `parallel` and `kernels` regions. Each of these directives will be detailed in the sections that follow.

The Kernels Construct

The `kernels` construct identifies a region of code that may contain parallelism, but relies on the automatic parallelization capabilities of the compiler to analyze the region, identify which loops are safe to parallelize, and then accelerate those loops. Developers will little or no parallel programming experience, or those working on functions containing many loop nests that might be parallelized will find the kernels directive a good starting place for OpenACC acceleration. The code below demonstrates the use of `kernels` in both C/C++ and Fortran.

```
#pragma acc kernels
{
  for (i=0; i<N; i++)
  {
    y[i] = 0.0f;
    x[i] = (float)(i+1);
  }

  for (i=0; i<N; i++)
```

```
    {
      y[i] = 2.0f * x[i] + y[i];
    }
  }
```

```
!$acc kernels
do i=1,N
  y(i) = 0
  x(i) = i
enddo

do i=1,N
  y(i) = 2.0 * x(i) + y(i)
enddo
!$acc end kernels
```

In this example the code is initializing two arrays and then performing a simple calculation on them. Notice that we have identified a block of code, using curly braces in C and starting and ending directives in Fortran, that contains two candidate loops for acceleration. The compiler will analyze these loops for data independence and parallelize both loops by generating an accelerator *kernel* for each. The compiler is given complete freedom to determine how best to map the parallelism available in these loops to the hardware, meaning that we will be able to use this same code regardless of the accelerator we are building for. The compiler will use its own knowledge of the target accelerator to choose the best path for acceleration. One caution about the `kernels` directive, however, is that if the compiler cannot be certain that a loop is data independent, it will not parallelize the loop. Common reasons for why a compiler may misidentify a loop as non-parallel will be discussed in a later section.

The Parallel Construct

The `parallel` construct identifies a region of code that will be parallelized across OpenACC *gangs*. By itself a `parallel` region is of limited use, but when paired with the `loop` directive (discussed in more detail later) the compiler will generate a parallel version of the loop for the accelerator. These two directives can, and most often are, combined into a single `parallel loop` directive. By placing this directive on a loop the programmer asserts that the affected loop is safe to parallelize and allows the compiler to select how to schedule the loop iterations on the target accelerator. The code below demonstrates the use of the `parallel loop` combined directive in both C/C++ and Fortran.

```
#pragma acc parallel loop
  for (i=0; i<N; i++)
  {
    y[i] = 0.0f;
    x[i] = (float)(i+1);
  }

#pragma acc parallel loop
  for (i=0; i<N; i++)
  {
    y[i] = 2.0f * x[i] + y[i];
  }
```

```
    !$acc parallel loop
    do i=1,N
      y(i) = 0
      x(i) = i
    enddo

    !$acc parallel loop
    do i=1,N
      y(i) = 2.0 * x(i) + y(i)
    enddo
```

Notice that, unlike the `kernels` directive, each loop needs to be explicitly decorated with `parallel loop` directives. This is because the `parallel` construct relies on the programmer to identify the parallelism in the code rather than performing its own compiler analysis of the loops. In this case, the programmer is only identifying the availability of parallelism, but still leaving the decision of how to map that parallelism to the accelerator to the compiler's knowledge about the device. This is a key feature that differentiates OpenACC from other, similar programming models. The programmer identifies the parallelism without dictating to the compiler how to exploit that parallelism. This means that OpenACC code will be portable to devices other than the device on which the code is being developed, because details about how to parallelize the code are left to compiler knowledge rather than being hard-coded into the source.

Differences Between Parallel and Kernels

One of the biggest points of confusion for new OpenACC programmers is why the specification has both the `parallel` and `kernels` directives, which appear to do the same thing. While they are very closely related there are subtle differences between them. The `kernels` construct gives the compiler maximum leeway to parallelize and optimize the code how it sees fit for the target accelerator, but also relies most heavily on the compiler's ability to automatically parallelize the code. As a result, the programmer may see differences in what different compilers are able to parallelize and how they do so. The `parallel loop` directive is an assertion by the programmer that it is both safe and desirable to parallelize the affected loop. This relies on the programmer to have correctly identified parallelism in the code and remove anything in the code that may be unsafe to parallelize. If the programmer asserts incorrectly that the loop may be parallelized then the resulting application may produce incorrect results.

To put things another way: the `kernels` construct may be thought of as a hint to the compiler of where it should look for parallelism while the `parallel` directive is an assertion to the compiler of where there is parallelism.

An important thing to note about the `kernels` construct is that the compiler will analyze the code and only parallelize when it is certain that it is safe to do so. In some cases the compiler may not have enough information at compile time to determine whether a loop is safe the parallelize, in which case it will not parallelize the loop, even if the programmer can clearly see that the loop is safely parallel. For example, in the case of C/C++ code, where arrays are passed into functions as pointers, the compiler may not always be able to determine that two arrays do not share the same memory, otherwise known as *pointer aliasing*. If the compiler cannot know that two pointers are not aliased it will not be able to parallelize a loop that accesses those arrays.

memory hazard!

Best Practice: C programmers should use the `restrict` keyword (or the `__restrict` decorator in C++) whenever possible to inform the compiler that the pointers are not aliased, which will frequently give the compiler enough information to then parallelize loops that it would not have otherwise. In addition to the `restrict` keyword, declaring constant variables using the `const` keyword may allow the compiler to use a read-only memory for that variable if such a memory exists on the accelerator. Use of `const` and `restrict` is a good programming practice in general, as it gives the compiler additional information that can be used when optimizing the code.

Fortran programmers should also note that an OpenACC compiler will parallelize Fortran array syntax that is contained in a `kernels` construct. When using `parallel` instead, it will be necessary to explicitly introduce loops over the elements of the arrays.

One more notable benefit that the `kernels` construct provides is that if data is moved to the device for use in loops contained in the region, that data will remain on the device for the full extent of the region, or until it is needed again on the host within that region. This means that if multiple loops access the same data it will only be copied to the accelerator once. When `parallel loop` is used on two subsequent loops that access the same data a compiler may or may not copy the data back and forth between the host and the device between the two loops. In the examples shown in the previous section the compiler generates implicit data movement for both parallel loops, but only generates data movement once for the `kernels` approach, which may result in less data motion by default. This difference will be revisited in the case study later in this chapter.

For more information on the differences between the `kernels` and `parallel` directives, please see [http://www.pgroup.com/lit/articles/insider/v4n2a1.htm].

At this point many programmers will be left wondering which directive they should use in their code. More experienced parallel programmers, who may have already identified parallel loops within their code, will likely find the `parallel loop` approach more desirable. Programmers with less parallel programming experience or whose code contains a large number of loops that need to be analyzed may find the `kernels` approach much simpler, as it puts more of the burden on the compiler. Both approaches have advantages, so new OpenACC programmers should determine for themselves which approach is a better fit for them. A programmer may even choose to use `kernels` in one part of the code, but `parallel` in another if it makes sense to do so.

Note: For the remainder of the document the phrase *parallel region* will be used to describe either a `parallel` or `kernels` region. When refering to the `parallel` construct, a terminal font will be used, as shown in this sentence.

The Loop Construct

The `loop` construct gives the compiler additional information about the very next loop in the source code. The `loop` directive was shown above in connection with the `parallel` directive, although it is also valid with `kernels`. Loop clauses come in two forms: clauses for correctness and clauses for optimization. This chapter will only discuss the two correctness clauses and a later chapter will discuss optimization clauses.

private

The private clause specifies that each loop iteration requires its own copy of the listed variables. For example, if each loop contains a small, temporary array named `tmp` that it uses during its calculation, then this variable must be made private to each loop iteration in order to ensure correct results. If `tmp` is not declared private, then threads executing different loop iterations may access this shared `tmp` variable in unpredictable ways, resulting in a race condition and potentially incorrect results. Below is the synax for the `private` clause.

```
private(variable)
```

There are a few special cases that must be understood about scalar variables within loops. First, loop iterators will be privatized by default, so they do not need to be listed as private. Second, unless otherwise specified, any scalar accessed within a parallel loop will be made *first private* by default, meaning a private copy will be made of the variable for each loop iteration and it will be initialized with the value of that scalar upon entering the region. Finally, any variables (scalar or not) that are declared within a loop in C or C++ will be made private to the iterations of that loop by default.

Note: The `parallel` construct also has a `private` clause which will privatize the listed variables for each gang in the parallel region.

reduction

The `reduction` clause works similarly to the `private` clause in that a private copy of the affected variable is generated for each loop iteration, but `reduction` goes a step further to reduce all of those private copies into one final result, which is returned from the region. For example, the maximum of all private copies of the variable may be required or perhaps the sum. A reduction may only be specified on a scalar variable and only common, specified operations can be performed, such as `+`, `*`, `min`, `max`, and various bitwise operations (see the OpenACC specification for a complete list). The format of the reduction clause is as follows, where *operator* should be replaced with the operation of interest and *variable* should be replaced with the variable being reduced:

```
reduction(operator:variable)
```

An example of using the `reduction` clause will come in the case study below.

Routine Directive

Function or subroutine calls within parallel loops can be problematic for compilers, since it's not always possible for the compiler to see all of the loops at one time. OpenACC 1.0 compilers were forced to either inline all routines called within parallel regions or not parallelize loops containing routine calls at all. OpenACC 2.0 introduced the `routine` directive to address this shortcoming. The `routine` directive gives the compiler the necessary information about the function or subroutine and the loops it contains in order to parallelize the calling parallel region. The routine directive must be added to a function definition informing the compiler of the level of parallelism used within the routine. OpenACC's *levels of parallelism* will be discussed in a later section.

C++ Class Functions

When operating on C++ classes, it's frequently necessary to call class functions from within parallel regions. The example below shows a C++ class `float3` that contains 3 floating point values and has a `set` function that is used to set the values of its `x`, `y`, and `z` members to that of another instance of `float3`. In order for this to work from within a parallel region, the `set` function is declared as an OpenACC routine using the `acc routine` directive. Since we know that it will be called by each iteration of a parallel loop, it's declared a `seq` (or *sequential*) routine.

```
class float3 {
   public:
    float x,y,z;

   #pragma acc routine seq
   void set(const float3 *f) {
    x=f->x;
    y=f->y;
```

```
        z=f->z;
      }
    };
```

Case Study - Parallelize

In the last chapter we identified the two loop nests within the convergence loop as the most time consuming parts of our application. Additionally we looked at the loops and were able to determine that the outer, convergence loop is not parallel, but the two loops nested within are safe to parallelize. In this chapter we will accelerate those loop nests with OpenACC using the directives discussed earlier in this chapter. To further emphasize the similarities and differences between `parallel` and `kernels` directives, we will accelerate the loops using both and discuss the differences.

Parallel Loop

We previously identified the available parallelism in our code, now we will use the `parallel loop` directive to accelerate the loops that we identified. Since we know that the two, doubly-nested sets of loops are parallel, simply add a `parallel loop` directive above each of them. This will inform the compiler that the outer of the two loops is safely parallel. Some compilers will additionally analyze the inner loop and determine that it is also parallel, but to be certain we will also add a `loop` directive around the inner loops.

There is one more subtlety to accelerating the loops in this example: we are attempting to calculate the maximum value for the variable `error`. As discussed above, this is considered a *reduction* since we are reducing from all possible values for `error` down to just the single maximum. This means that it is necessary to indicate a reduction on the first loop nest (the one that calculates `error`).

Best Practice: Some compilers will detect the reduction on `error` and implicitly insert the `reduction` clause, but for maximum portability the programmer should always indicate reductions in the code.

At this point the code looks like the examples below.

```
    while ( error > tol && iter < iter_max )
    {
      error = 0.0;

      #pragma acc parallel loop reduction(max:error)
      for( int j = 1; j < n-1; j++)
      {
        #pragma acc loop reduction(max:error)
        for( int i = 1; i < m-1; i++ )
        {
          A[j][i] = 0.25 * ( Anew[j][i+1] + Anew[j][i-1]
                           + Anew[j-1][i] + Anew[j+1][i]);
          error = fmax( error, fabs(A[j][i] - Anew[j][i]));
        }
      }

      #pragma acc parallel loop
      for( int j = 1; j < n-1; j++)
      {
        #pragma acc loop
        for( int i = 1; i < m-1; i++ )
        {
```

```
        A[j][i] = Anew[j][i];
      }
    }

    if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

    iter++;
  }
```

```
  do while ( error .gt. tol .and. iter .lt. iter_max )
    error=0.0_fp_kind

    !$acc parallel loop reduction(max:error)
    do j=1,m-2
      !$acc loop reduction(max:error)
      do i=1,n-2
        A(i,j) = 0.25_fp_kind * ( Anew(i+1,j  ) + Anew(i-1,j  ) + &
                                  Anew(i  ,j-1) + Anew(i  ,j+1) )
        error = max( error, abs(A(i,j) - Anew(i,j)) )
      end do
    end do

    !$acc parallel loop
    do j=1,m-2
      !$acc loop
      do i=1,n-2
        A(i,j) = Anew(i,j)
      end do
    end do

    if(mod(iter,100).eq.0 ) write(*,'(i5,f10.6)'), iter, error
    iter = iter + 1
  end do
```

Best Practice: Most OpenACC compilers will accept only the `parallel loop` directive on the `j` loops and detect for themselves that the `i` loop can also be parallelized without needing the `loop` directives on the `i` loops. By placing a `loop` directive on each loop that I know can be parallelized the programmer ensures that the compiler will understand that the loop is safe the parallelize. When used within a `parallel` region, the `loop` directive asserts that the loop iterations are independent of each other and are safe the parallelize and should be used to provide the compiler as much information about the loops as possible.

Building the above code using the PGI compiler (version 15.5) produces the following compiler feedback (shown for C, but the Fortran output is similar).

```
$ pgcc -acc -ta=tesla -Minfo=accel laplace2d-parallel.c
main:
     56, Accelerator kernel generated
         56, Max reduction generated for error
         57, #pragma acc loop gang /* blockIdx.x */
         60, #pragma acc loop vector(128) /* threadIdx.x */
             Max reduction generated for error
     56, Generating copyout(Anew[1:4094][1:4094])
```

```
         Generating copyin(A[:][:])
         Generating Tesla code
     60, Loop is parallelizable
     68, Accelerator kernel generated
         69, #pragma acc loop gang /* blockIdx.x */
         72, #pragma acc loop vector(128) /* threadIdx.x */
     68, Generating copyin(Anew[1:4094][1:4094])
         Generating copyout(A[1:4094][1:4094])
         Generating Tesla code
     72, Loop is parallelizable
```

Analyzing the compiler feedback gives the programmer the ability to ensure that the compiler is producing the expected results and fix any problems if it's not. In the output above we see that accelerator kernels were generated for the two loops that were identified (at lines 58 and 71, in the compiled source file) and that the compiler automatically generated data movement, which will be discussed in more detail in the next chapter.

Other clauses to the `loop` directive that may further benefit the performance of the resulting code will be discussed in a later chapter.

Kernels

Using the `kernels` construct to accelerate the loops we've identified requires inserting just one directive in the code and allowing the compiler to perform the parallel analysis. Adding a `kernels` construct around the two computational loop nests results in the following code.

```
    while ( error > tol && iter < iter_max )
    {
      error = 0.0;

      #pragma acc kernels
      {
        for( int j = 1; j < n-1; j++)
        {
          for( int i = 1; i < m-1; i++ )
          {
            A[j][i] = 0.25 * ( Anew[j][i+1] + Anew[j][i-1]
                                + Anew[j-1][i] + Anew[j+1][i]);
            error = fmax( error, fabs(A[j][i] - Anew[j][i]));
          }
        }

        for( int j = 1; j < n-1; j++)
        {
          for( int i = 1; i < m-1; i++ )
          {
            A[j][i] = Anew[j][i];
          }
        }
      }

      if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

      iter++;
    }
    do while ( error .gt. tol .and. iter .lt. iter_max )
      error=0.0_fp_kind

      !$acc kernels
      do j=1,m-2
        do i=1,n-2
          A(i,j) = 0.25_fp_kind * ( Anew(i+1,j  ) + Anew(i-1,j  ) + &
                                    Anew(i  ,j-1) + Anew(i  ,j+1) )
          error = max( error, abs(A(i,j) - Anew(i,j)) )
        end do
      end do

      do j=1,m-2
        do i=1,n-2
          A(i,j) = Anew(i,j)
        end do
      end do
      !$acc end kernels

      if(mod(iter,100).eq.0 ) write(*,'(i5,f10.6)'), iter, error
      iter = iter + 1
    end do
```

The above code demonstrates some of the power that the `kernels` construct provides, since the compiler will analyze the code and identify both loop nests as parallel and it will automatically discover the reduction on the `error` variable without programmer intervention. An OpenACC compiler will likely discover not only that the outer loops are parallel, but also the inner loops, resulting in more available parallelism with fewer directives than the `parallel loop` approach. Had the programmer put the `kernels` construct around the convergence loop, which we have already determined is not parallel, the compiler likely would not have found any available parallelism. Even with the `kernels` directive it is necessary for the programmer to do some amount of analysis to determine where parallelism may be found.

Taking a look at the compiler output points to some more subtle differences between the two approaches.

```
$ pgcc -acc -ta=tesla -Minfo=accel laplace2d-kernels.c
main:
     56, Generating copyout(Anew[1:4094][1:4094])
         Generating copyin(A[:][:])
         Generating copyout(A[1:4094][1:4094])
         Generating Tesla code
     58, Loop is parallelizable
     60, Loop is parallelizable
         Accelerator kernel generated
         58, #pragma acc loop gang /* blockIdx.y */
         60, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         64, Max reduction generated for error
     68, Loop is parallelizable
     70, Loop is parallelizable
         Accelerator kernel generated
         68, #pragma acc loop gang /* blockIdx.y */
         70, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

The first thing to notice from the above output is that the compiler correctly identified all four loops as being parallelizable and generated kernels from those loops. Also notice that the compiler only generated implicit data movement directives at line 54 (the beginning of the `kernels` region), rather than at the beginning of each `parallel loop`. This means that the resulting code should perform fewer copies between host and device memory in this version than the version from the previous section. A more subtle difference between the output is that the compiler chose a different loop decomposition scheme (as is evident by the implicit `acc loop` directives in the compiler output) than the parallel loop because `kernels` allowed it to do so. More details on how to interpret this decomposition feedback and how to change the behavior will be discussed in a later chapter.

At this point we have expressed all of the parallelism in the example code and the compiler has parallelized it for an accelerator device. Analyzing the performance of this code may yield surprising results on some accelerators, however. The results below demonstrate the performance of this code on 1 - 8 CPU threads on a modern CPU at the ime of publication and an NVIDIA Tesla K40 GPU using both implementations above. The *y axis* for figure 3.1 is execution time in seconds, so smaller is better. For the two OpenACC versions, the bar is divided by time transferring data between the host and device, time executing on the device, and other time.

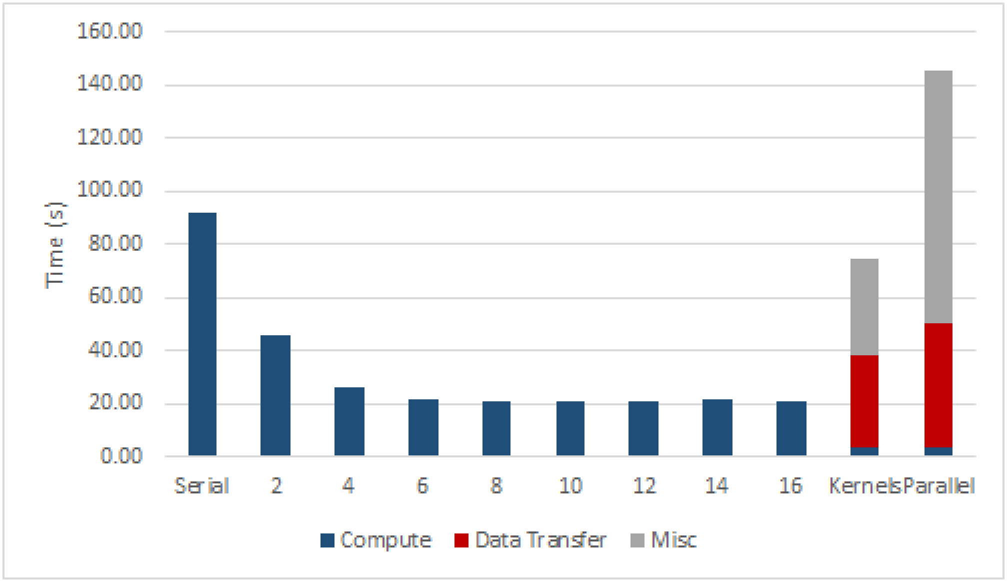


Figure 3.1: Jacobi Iteration Performance - Step 1

Notice that the performance of this code improves as CPU threads are added to the calcuation, but the OpenACC versions perform poorly compared to the CPU baseline. The OpenACC `kernels` version performs slightly better than the serial version, but the `parallel loop` case performs dramaticaly worse than even the slowest CPU version. Further performance analysis is necessary to identify the source of this slowdown. This analysis has already been applied to the graph above, which breaks down time spent computing the solution, copying data to and from the accelerator, and miscelaneous time, which includes various overheads involved in scheduling data transfers and computation.

A variety of tools are available for performing this analysis, but since this case study was compiled for an NVIDIA GPU, the NVIDIA Visual profiler will be used to understand the application peformance. The screenshot in figure 3.2 shows NVIDIA Visual Profiler for *2* iterations of the convergence loop in the `parallel loop` version of the code.

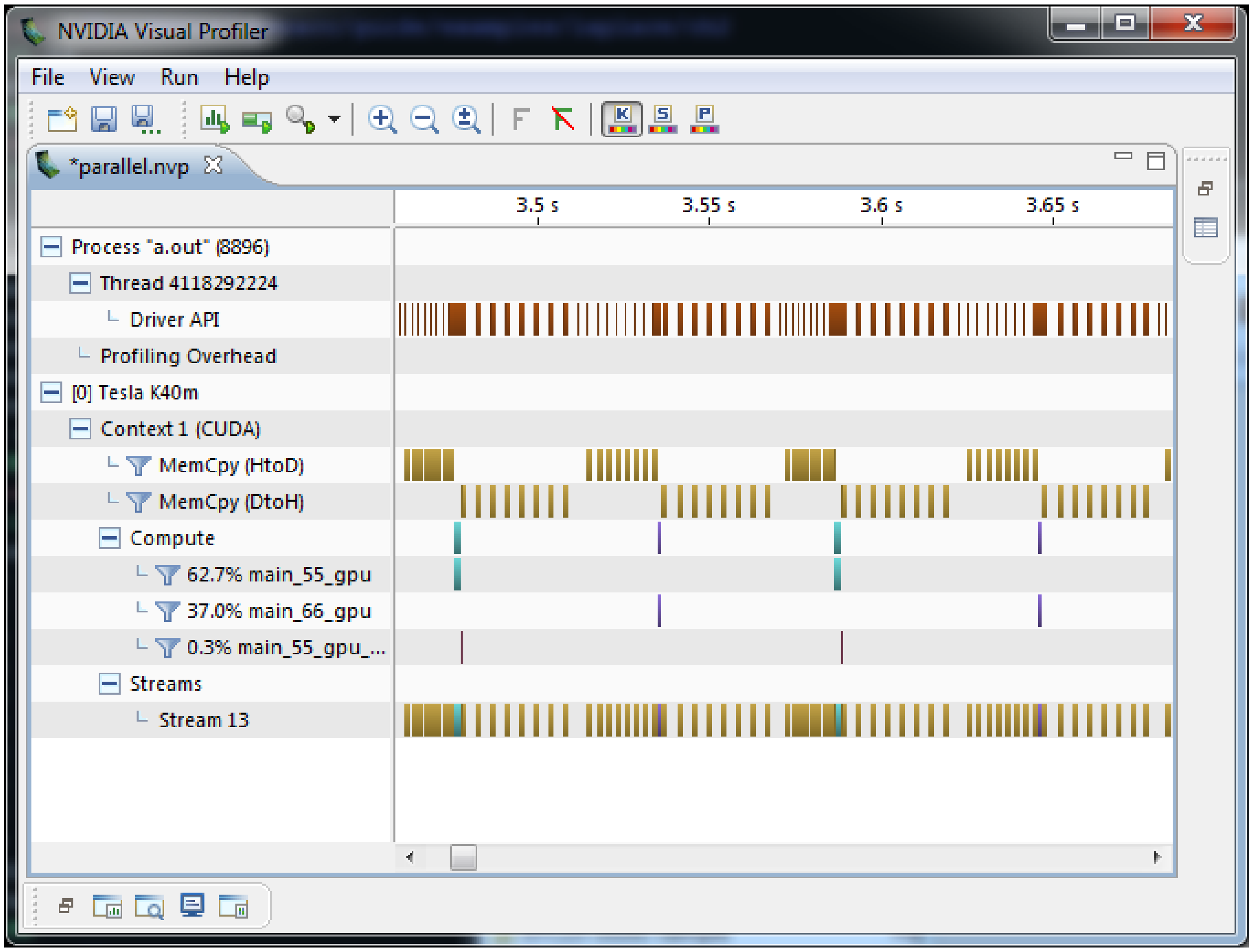


Figure 3.2: Screenshot of NVIDIA Visual Profiler on 2 steps of the Jacobi Iteration showing a high amount of data transfer compared to computation.

Since the test machine has two distinct memory spaces, one for the CPU and one for the GPU, it's necessary to copy data between the two memories. In this screenshot, the tool represents data transfers using the tan colored boxes in the two *MemCpy* rows and the computation time in the green and purple boxes in the rows below *Compute*. It should be obvious from the timeline displayed that significantly more time is being spent copying data to and from the accelerator before and after each compute kernel than actually computing on the device. In fact, the majority of the time is spent either in memory copies or in overhead incurred by the runtime scheduling memory copeis. In the next chapter we will fix this inefficiency, but first, why does the `kernels` version outperform the `parallel loop` version?

When an OpenACC compiler parallelizes a region of code it must analyze the data that is needed within that region and copy it to and from the accelerator if necessary. This analysis is done at a per-region level and will typically default to copying arrays used on the accelerator both to and from the device at the beginning and end of the region respectively. Since the `parallel loop` version has two compute regions, as opposed to only one in the `kernels` version, data is copied back and forth between the two regions. As a result, the copy and overhead times are roughly twice that of the `kernels` region, although the compute kernel times are roughly the same.

Atomic Operations

When one or more loop iterations need to access an element in memory at the same time data races can occur. For instance, if one loop iteration is modifying the value contained in a variable and another is trying to read from the same variable in parallel, different results may occur depending on which iteration occurs first. In serial programs, the sequential loops ensure that the variable will be modified and read in a predictable order, but parallel programs don't make guarantees that a particular loop iteration will happen before anoter. In simple cases, such as finding a sum, maximum, or minimum value, a reduction operation will ensure correctness. For more complex operations, the `atomic` directive will ensure that no two threads can attempt to perfom the contained operation simultaneously. Use of atomics is sometimes a necessary part of parallelization to ensure correctness.

The `atomic` directive accepts one of four clauses to declare the type of operation contained within the region. The `read` operation ensures that no two loop iterations will read from the region at the same time. The `write` operation will ensure that no two iterations with write to the region at the same time. An `update` operation is a combined read and write. Finally a `capture` operation performs an update, but saves the value calculated in that region to use in the code that follows. If no clause is given then an update operation will occur.

Atomic Example

A histogram is a common technique for counting up how many times values occur from an input set according to their value. Figure __ shows a histogram that counts the number of times numbers fall within particular ranges. The example code below loops through a series of integer numbers of a known range and counts the occurances of each number in that range. Since each number in the range can occur multiple times, we need to ensure that each element in the histogram array is updated atomically. The code below demonstrates using the `atomic` directive to generate a histogram.

```
#pragma acc parallel loop
for(int i=0;i<HN;i++)
  h[i]=0;

#pragma acc parallel loop
for(int i=0;i<N;i++) {
  #pragma acc atomic update
```

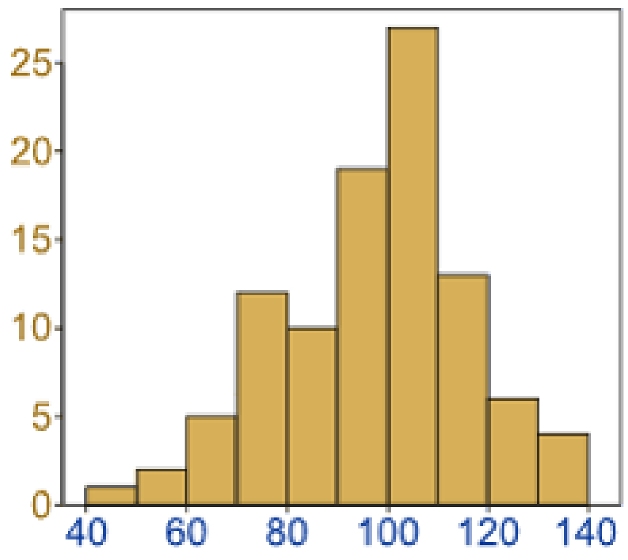


Figure 3.3: A histogram of number distribution.

```
    h[a[i]]+=1;
  }
```

```
!$acc kernels
h(:) = 0
!$acc end kernels
!$acc parallel loop
do i=1,N
  !$acc atomic
  h(a(i)) = h(a(i)) + 1
enddo
!$acc end parallel loop
```

Notice that updates to the histogram array `h` are performed atomically. Because we are incrementing the value of the array element, an update operation is used to read the value, modify it, and then write it back.

Chapter 4

Optimize Data Locality

At the end of the previous chapter we saw that although we've moved the most compute intensive parts of the application to the accelerator, sometimes the process of copying data from the host to the accelerator and back will be more costly than the computation itself. This is because it's difficult for a compler to determine when (or if) the data will be needed in the future, so it must be cautious and ensure that the data will be copied in case it's needed. To improve upon this, we'll exploit the *data locality* of the application. Data locality means that data used in device or host memory should remain local to that memory for as long as it's needed. This idea is sometimes referred to as optimizing data reuse or optimizing away unnecessary data copies between the host and device memories. However you think of it, providing the compiler with the information necessary to only relocate data when it needs to do so is frequently the key to success with OpenACC.

After expressing the parallelism of a program's important regions it's frequently necessary to provide the compiler with additional information about the locality of the data used by the parallel regions. As noted in the previous section, a compiler will take a cautious approach to data movement, always copying data that may be required, so that the program will still produce correct results. A programmer will have knowledge of what data is really needed and when it will be needed. The programmer will also have knowledge of how data may be shared between two functions, something that is difficult for a compiler to determine. Even when does not immediately know how best to optimize data motion, profiling tools may help the programmer identify when excess data movement occurs, as will be shown in the case study at the end of this chapter.

The next step in the acceleration process is to provide the compiler with additional information about data locality to maximize reuse of data on the device and minimize data transfers. It is after this step that most applications will observe the benefit of OpenACC acceleration. This step will be primarily beneficial on machine where the host and device have separate memories.

Data Regions

The `data` construct facilitates the sharing of data between multiple parallel regions. A data region may be added around one or more parallel regions in the same function or may be placed at a higher level in the program call tree to enable data to be shared between regions in multiple functions. The `data` construct is a structured construct, meaning that it must begin and end in the same scope (such as the same function or subroutine). A later section will discuss how to handle cases where a structured construct is not useful. A `data` region may be added to the earlier `parallel loop` example to enable data to be shared between both loop nests as follows.

```
#pragma acc data
{
  #pragma acc parallel loop
    for (i=0; i<N; i++)
    {
      y[i] = 0.0f;
      x[i] = (float)(i+1);
    }

  #pragma acc parallel loop
    for (i=0; i<N; i++)
    {
      y[i] = 2.0f * x[i] + y[i];
    }
}
```

```
!$acc data
!$acc parallel loop
do i=1,N
  y(i) = 0
  x(i) = i
enddo

!$acc parallel loop
do i=1,N
  y(i) = 2.0 * x(i) + y(i)
enddo
!$acc end data
```

The `data` region in the above examples enables the `x` and `y` arrays to be reused between the two `parallel` regions. This will remove any data copies that happen between the two regions, but it still does not guarantee optimal data movement. In order to provide the information necessary to perform optimal data movement, the programmer can add data clauses to the `data` region.

Note: An implicit data region is created by each `parallel` and `kernels` region.

Data Clauses

Data clauses give the programmer additional control over how and when data is created on and copied to or from the device. These clauses may be added to any `data`, `parallel`, or `kernels` construct to inform the compiler of the data needs of that region of code. The data directives, along with a brief description of their meanings, follow.

- `copy` - Create space for the listed variables on the device, initialize the variable by copying data to the device at the beginning of the region, copy the results back to the host at the end of the region, and finally release the space on the device when done. like CUDA
- `copyin` - Create space for the listed variables on the device, initialize the variable by copying data to the device at the beginning of the region, and release the space on the device when done without copying the data back the the host. like the don't return in R

for device generated stuff

- **copyout** - Create space for the listed variables on the device but do not initialize them. At the end of the region, copy the results back to the host and release the space on the device.
- **create** - Create space for the listed variables and release it at the end of the region, but do not copy to or from the device. maybe for temporary local variables?
- **present** - The listed variables are already present on the device, so no further action needs to be taken. This is most frequently used when a data region exists in a higher-level routine.
- **deviceptr** - The listed variables use device memory that has been managed outside of OpenACC, therefore the variables should be used on the device without any address translation. This clause is generally used when OpenACC is mixed with another programming model, as will be discussed in the interoperability chapter.

In addition to these data clauses, OpenACC 1.0 and 2.0 provide `present_or_*` clauses (`present_or_copy`, for instance) that inform the compiler to check whether the variable is already present on the device; if it is present, use that existing copy of the data, if it is not, perform the action listed. These routines are frequently abbreviated, like `pcopyin` instead of `present_or_copyin`. In an upcoming OpenACC specification the behavior of all data directives will be *present or*, so programmers should begin writing their applications using these directives to ensure correctness with future OpenACC specifications. This change will simplify data reuse for the programmer.

Shaping Arrays

Sometimes a compiler will need some extra help determining the size and shape of arrays used in parallel or data regions. For the most part, Fortran programmers can rely on the self-describing nature of Fortran arrays, but C/C++ programmers will frequently need to give additional information to the compiler so that it will know how large an array to allocate on the device and how much data needs to be copied. To give this information the programmer adds a *shape* specification to the data clauses.

In C/C++ the shape of an array is described as `x[start:count]` where *start* is the first element to be copied and *count* is the number of elements to copy. If the first element is 0, then it may be left off.

In Fortran the shape of an array is described as `x(start:end)` where *start* is the first element to be copied and *end* is the last element to be copied. If *start* is the beginning of the array or *end* is the end of the array, they may be left off.

Array shaping is frequently necessary in C/C++ codes when the OpenACC appears inside of function calls or the arrays are dynamically allocated, since the shape of the array will not be known at compile time. Shaping is also useful when only a part of the array needs to be stored on the device.

As an example of array shaping, the code below modifies the previous example by adding shape information to each of the arrays.

```
#pragma acc data pcreate(x[0:N]) pcopyout(y[0:N])
{
  #pragma acc parallel loop
    for (i=0; i<N; i++)
    {
      y[i] = 0.0f;
      x[i] = (float)(i+1);
    }

  #pragma acc parallel loop
    for (i=0; i<N; i++)
    {
      y[i] = 2.0f * x[i] + y[i];
    }
}
```

```
!$acc data pcreate(x(1:N)) pcopyout(y(1:N))
!$acc parallel loop
do i=1,N
  y(i) = 0
  x(i) = i
enddo

!$acc parallel loop
do i=1,N
  y(i) = 2.0 * x(i) + y(i)
enddo
!$acc end data
```

With these data clauses it is possible to further improve the example shown above by informing the compiler how and when it should perform data transfers. In this simple example above, the programmer knows that both `x` and `y` will be populated with data on the device, so neither will need to be copied to the device, but the results of `y` are significant, so it will need to be copied back to the host at the end of the calculation. The code below demonstrates using the `pcreate` and `pcopyout` directives to describe exactly this data locality to the compiler.

```
#pragma acc data pcreate(x) pcopyout(y)
{
  #pragma acc parallel loop
    for (i=0; i<N; i++)
    {
      y[i] = 0.0f;
      x[i] = (float)(i+1);
    }

  #pragma acc parallel loop
    for (i=0; i<N; i++)
    {
      y[i] = 2.0f * x[i] + y[i];
    }
}
```

```
!$acc data pcreate(x) pcopyout(y)
!$acc parallel loop
do i=1,N
  y(i) = 0
  x(i) = i
enddo

```

```
    !$acc parallel loop
    do i=1,N
      y(i) = 2.0 * x(i) + y(i)
    enddo
    !$acc end data
```

Unstructured Data Lifetimes

While structured data regions are generally sufficient for optimizing the data locality in a program, they are not sufficient for some programs, particularly those using Object Oriented coding practices. For example, in a C++ class data is frequently allocated in a class constructor, deallocated in the destructor, and cannot be accessed outside of the class. This makes using structured data regions impossible because there is no single, structured scope where the construct can be placed. For these situations OpenACC 2.0 introduced unstructured data lifetimes. The `enter data` and `exit data` directives can be used to identify precisely when data should be allocated and deallocated on the device.

The `enter data` directive accepts the `create` and `copyin` data clauses and may be used to specify when data should be created on the device.

The `exit data` directive accepts the `copyout` and a special `delete` data clause to specify when data should be removed from the device.

Please note that multiple `enter data` directives may place an array on the device, but when any `exit data` directive removes it from the device it will be immediately removed, regardless of how many `enter data` regions reference it.

C++ Class Data

C++ class data is one of the primary reasons that unstructured data lifetimes were added to OpenACC. As described above, the encapsulation provided by classes makes it impossible to use a structured `data` region to control the locality of the class data. Programmers may choose to use the unstructured data lifetime directives or the OpenACC API to control data locality within a C++ class. Use of the directives is preferable, since they will be safely ignored by non-OpenACC compilers, but the API is also available for times when the directives are not expressive enough to meet the needs of the programmer. The API will not be discussed in this guide, but is well-documented on the OpenACC website.

The example below shows a simple C++ class that has a constructor, a destructor, and a copy constructor. The data management of these routines has been handled using OpenACC directives.

```
template <class ctype> class Data
{
  private:
    /// Length of the data array
    int len;
    /// Data array
    ctype *arr;

  public:
    /// Class constructor
    Data(int length)
    {
      len = length;
      arr = new ctype[len];
#pragma acc enter data copyin(this)
#pragma acc enter data create(arr[0:len])
    }

    /// Copy constructor
    Data(const Data<ctype> &d)
    {
      len = d.len;
      arr = new ctype[len];
#pragma acc enter data copyin(this)
#pragma acc enter data create(arr[0:len])
#pragma acc parallel loop present(arr[0:len],d)
      for(int i = 0; i < len; i++)
        arr[i] = d.arr[i];
    }

    /// Class destructor
    ~Data()
    {
#pragma acc exit data delete(arr)
#pragma acc exit data delete(this)
      delete arr;
      len = 0;
    }
};
```

Notice that an `enter data` directive is added to the class constructor to handle creating space for the class data on the device. In addition to the data array itself the `this` pointer is copied to the device. Copying the `this` pointer ensures that the scalar member `len`, which denotes the length of the data array `arr`, and the pointer `arr` are available on the accelerator as well as the host. It is important to place the `enter data` directive after the class data has been initialized. Similarly `exit data` directives are added to the destructor to handle cleaning up the device memory. It is important to place this directive before array members are freed, because once the host copies are freed the underlying pointer may become invalid, making it impossible to then free the device memory as well. For the same reason the `this` pointer should not be removed from the device until after all other memory has been released.

The copy constructor is a special case that is worth looking at on its own. The copy constructor will be responsible for allocating space on the device for the class that it is creating, but it will also rely on data that is managed by the class being copied. Since OpenACC does not currently provide a portable way to copy from one array to another, like a `memcpy` on the host, a loop is used to copy each individual element to from one array to the other. Because we know that the `Data` object passed in will also have its members on the device, we use a `present` clause on the `parallel loop` to inform the compiler that no data movement is necessary.

The same technique used in the class constructor and destructor above can be used in other programming languages as well. For instance, it's common practice in Fortran codes to have a subroutine that allocate and initialize all arrays contained within a module. Such a routine is a natural place to use an `enter data` region, as the allocation of both the host and device memory will appear within the same routine in the code. Placing `enter data` and `exit data` directives in close proximity to the usual allocation and deallocation of data within the code simplifies code maintenance.

Update Directive

Keeping data resident on the accelerator is often key to obtaining high performance, but sometimes it's necessary to copy data between host and device memories. The `update` directive provides a way to explicitly update the values of host or device memory with the values of the other. This can be thought of as syncrhonizing the contents of the two memories. The `update` directive accepts a `device` clause for copying data from the host to the device and a `self` directive for updating from the device to local memory, which is the host memory, except in the case of nested OpenACC regions. OpenACC 1.0 had a `host` clause, which is deprecated in OpenACC 2.0 and behaves the same as `self`. The `update` directive has other clauses and the more commonly used ones will be discussed in a later chapter.

As an example of the `update` directive, below are two routines that may be added to the above `Data` class to force a copy from host to device and device to host.

```
void update_host()
{
#pragma acc update self(arr[0:len])
  ;
}
void update_device()
{
#pragma acc update device(arr[0:len])
  ;
}
```

The update clauses accept an array shape, as already discussed in the data clauses section. Although the above example copies the entire `arr` array to or from the device, a partial array may also be provided to reduce the data transfer cost when only part of an array needs to be updated, such as when exchanging boundary conditions.

Best Practice: As noted earlier in the document, variables in an OpenACC code should always be thought of as a singular object, rather than a *host* copy and a *device* copy. Even when developing on a machine with a unified host and device memory it is important to include an `update` directive whenever accessing data from the host or device that was previous written to by the other, it is important to use an `update` directive to ensure correctness on all devices. For systems with distinct memories, the `update` will synchronize the values of the affected variable on the host and the device. On devices with a unified memory, the update will be ignored, incurring no performance penalty. In the example below, omiting the `update` on line 17 will produce different results on a unified and non-unified memory machine, making the code non-portable.

```
 1  for(int i=0; i<N; i++)
 2  {
 3    a[i] = 0;
 4    b[i] = 0;
 5  }
 6
 7  #pragma acc enter data copyin(a[0:N])
 8
 9  #pragma acc parallel loop
10  {
11    for(int i=0; i<N; i++)
12    {
13      a[i] = 1;
14    }
15  }
```

```

    #pragma acc update self(a[0:N])
    for(int i=0; i<N; i++)
    {
      b[i] = a[i];
    }

    #pragma acc exit data
```

Best Practice: Offload Inefficient Operations to Maintain Data Locality

Due to the high cost of PCIe data transfers on systems with distinct host and device memories, it's often beneficial to move sections of the application to the accelerator device, even when the code lacks sufficient parallelism to see direct benefit. The performance loss of running serial or code with a low degree of parallelism on a parallel accelerator is often less than the cost of transferring arrays back and forth between the two memories. A developer may use a `parallel` region with just 1 gang as a way to offload a serial section of the code to the accelerator. For instance, in the code below the first and last elements of the array are host elements that need to be set to zero. A `parallel` region (without a `loop`) is used to perform the parts that are serial.

```
    #pragma acc parallel loop
    for(i=1; i<(N-1); i++)
    {
      // calculate internal values
      A[i] = 1;
    }
    #pragma acc parallel
    {
      A[0]   = 0;
      A[N-1] = 0;
    }
```

```
    !$acc parallel loop
    do i=2,N-1
      ! calculate internal values
      A(i) = 1
    end do
    !$acc parallel
      A(1) = 0;
      A(N) = 0;
    !$acc end parallel
```

In the above example, the second `parallel` region will generate and launch a small kernel for setting the first and last elements. Small kernels generally do not run long enough to overcome the cost of a kernel launch on some offloading devices, such as GPUs. It's important that the data transfer saved by employing this technique is large enough to overcome the high cost of a kernel launch on some devices. Both the `parallel loop` and the second `parallel` region could be made asynchronous (discussed in a later chapter) to reduce the cost of the second kernel launch.

Note: Because the `kernels` directive instructs the compiler to search for parallelism, there is no similar technique for `kernels`, but the `parallel` approach above can be easily placed between `kernels` regions.

Case Study - Optimize Data Locality

By the end of the last chapter we had moved the main computational loops of our example code and, in doing so, introduced a significant amount of implicit data transfers. The performance profile for our code shows that for each iteration the `A` and `Anew` arrays are being copied back and forth between the *host* and *device*, four times for the `parallel loop` version and twice for the `kernels` version. Given that the values for these arrays are not needed until after the answer has converged, let's add a data region around the convergence loop. Additionally, we'll need to specify how the arrays should be managed by this data region. Both the initial and final values for the `A` array are required, so that array will require a `copy` data clause. The results in the `Anew` array, however, are only used within this section of code, so a `create` clause will be used for it. The resulting code is shown below.

Note: The changes required during this step are the same for both versions of the code, so only the `parallel loop` version will be shown.

```
    #pragma acc data copy(A[1:n][1:m]) create(Anew[n][m])
        while ( error > tol && iter < iter_max )
        {
            error = 0.0;

            #pragma acc parallel loop reduction(max:error)
            for( int j = 1; j < n-1; j++)
            {
                #pragma acc loop reduction(max:error)
                for( int i = 1; i < m-1; i++ )
                {
                    Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                        + A[j-1][i] + A[j+1][i]);
                    error = fmax( error, fabs(Anew[j][i] - A[j][i]));
                }
            }

           #pragma acc parallel loop
            for( int j = 1; j < n-1; j++)
            {
                #pragma acc loop
                for( int i = 1; i < m-1; i++ )
                {
                    A[j][i] = Anew[j][i];
                }
            }

            if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

            iter++;
        }
```

```fortran
    !$acc data copy(A) create(Anew)
    do while ( error .gt. tol .and. iter .lt. iter_max )
      error=0.0_fp_kind

      !$acc parallel loop reduction(max:error)
      do j=1,m-2
        !$acc loop reduction(max:error)
        do i=1,n-2
          A(i,j) = 0.25_fp_kind * ( Anew(i+1,j  ) + Anew(i-1,j  ) + &
                                    Anew(i  ,j-1) + Anew(i  ,j+1) )
          error = max( error, abs(A(i,j) - Anew(i,j)) )
        end do
      end do

      !$acc parallel loop
      do j=1,m-2
        !$acc loop
        do i=1,n-2
          A(i,j) = Anew(i,j)
        end do
      end do

      if(mod(iter,100).eq.0 ) write(*,'(i5,f10.6)'), iter, error
      iter = iter + 1
    end do
    !$acc end data
```

With this change, only the value computed for the maximum error, which is required by the convergence loop, is copied from the device every iteration. The `A` and `Anew` arrays will remain local to the device through the extent of this calculation. Using the Nvidia Visual Profiler again, we see that each data transfers now only occur at the beginning and end of the data region and that the time between each iterations is much less.

Looking at the final performance of this code, we see that the time for the OpenACC code on a GPU is now much faster than even the best threaded CPU code. Although only the `parallel loop` version is shown in the performance graph, the `kernels` version performs equally well once the `data` region has been added.

This ends the Jacobi Iteration case study. The simplicity of this implementation generally shows very good speed-ups with OpenACC, often leaving little potential for further improvement. The reader should feel encouraged, however, to revisit this code to see if further improvements are possible on the device of interest to them.

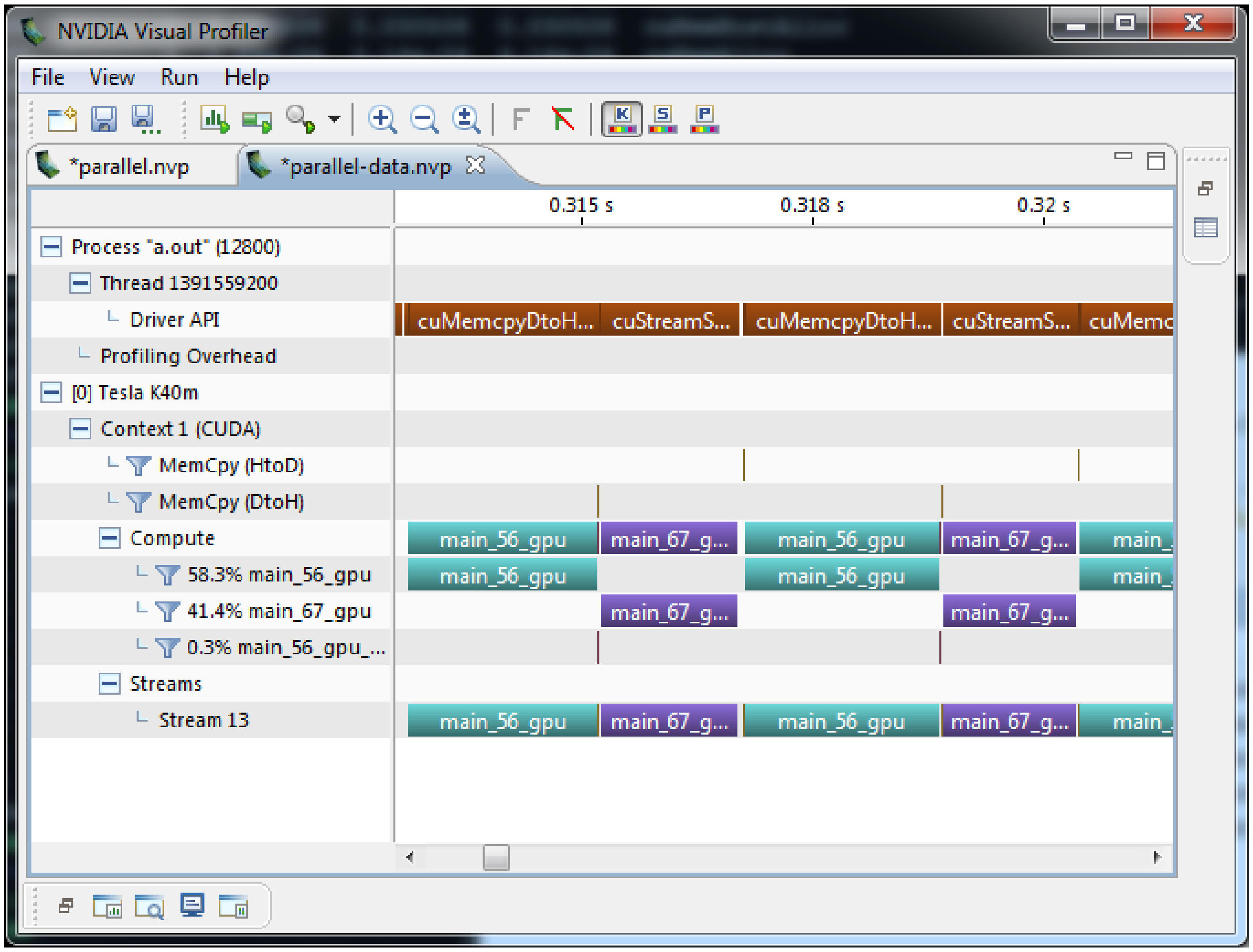


Figure 4.1: NVIDIA Visual Profiler showing 2 iterations of the Jacobi solver after adding the OpenACC data region.

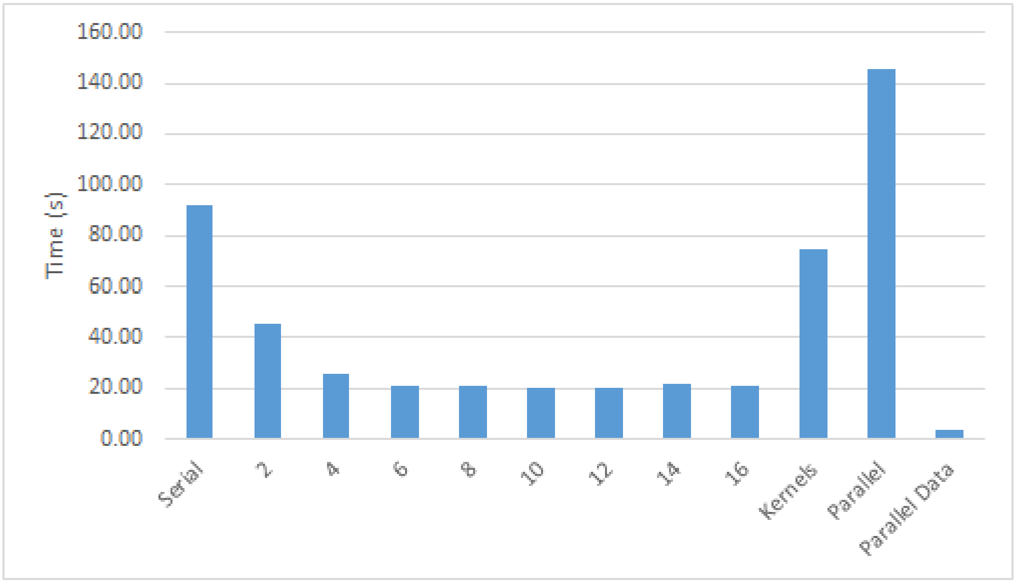


Figure 4.2: Runtime of Jacobi Iteration after adding OpenACC data region

Chapter 5

Optimize Loops

Once data locality has been expressed developers may wish to further tune the code for the hardware of interest. It's important to understand that the more loops are tuned for a particular type of hardware the less performance portable the code becomes to other architecures. If you're generally running on one particular accelerator, however, there may be some gains to be had by tuning how the loops are mapped to the underlying hardware.

It's tempting to begin tuning the loops before all of the data locality has been expressed in the code. Because PCIe copies are frequently the limiter to application performance on the current generation of accelerators the performance impact of tuning a particular loop may be too difficult to measure until data locality has been optimized. For this reason the best practice is to wait to optimize particular loops until after all of the data locality has been expressed in the code, reducing the PCIe transfer time to a minimum.

Efficient Loop Ordering

Before changing the way OpenACC maps loops onto the hardware of interest, the developer should examine the important loops to ensure that data arrays are being accessed in an efficient manner. Most modern hardware, be it a CPU with large caches and SIMD operations or a GPU with coalesced memory accesses and SIMT operations, favor accessing arrays in a *stride 1* manner. That is to say that each loop iteration accesses consecutive memory addresses. This is achieved by ensuring that the innermost loop of a loop nest iterates on the fastest varying array dimension and each successive loop outward accesses the next fastest varying dimension. Arranging loops in this increasing manner will frequently improve cache efficiency and improve vectorization on most architectures.

OpenACC's 3 Levels of Parallelism

OpenACC defines three levels of parallelism: *gang*, *worker*, and *vector*. Additionally execution may be marked as being sequential (*seq*). Vector parallelism has the finest granularity, with an individual instruction operating on multiple pieces of data (much like *SIMD* parallelism on a modern CPU or *SIMT* parallelism on a modern GPU). Vector operations are performed with a particular *vector length*, indicating how many data elements may be operated on with the same instruction. Gang parallelism is coarse-grained parallelism, where gangs work independently of each other and may not synchronize. Worker parallelism sits between vector and gang levels. A gang consists of 1 or more workers, each of which operates on a vector of some length. Within a gang the OpenACC model exposes a *cache* memory, which can be used by all workers and vectors within the gang, and it is legal to synchronize within a gang, although OpenACC does not expose synchronization to the user. Using these three levels of parallelism, plus sequential, a programmer can map the parallelism in the code to any device. OpenACC does not require the programmer to do this mapping explicitly, however. If the programmer chooses not to explicitly map loops to the device of interest the compiler will implicitly perform this mapping using what it knows about the target device. This makes OpenACC highly portable, since the same code may be mapped to any number of target devices. The more explicit mapping of parallelism the programmer adds to the code, however, the less portable they make the code to other architectures.

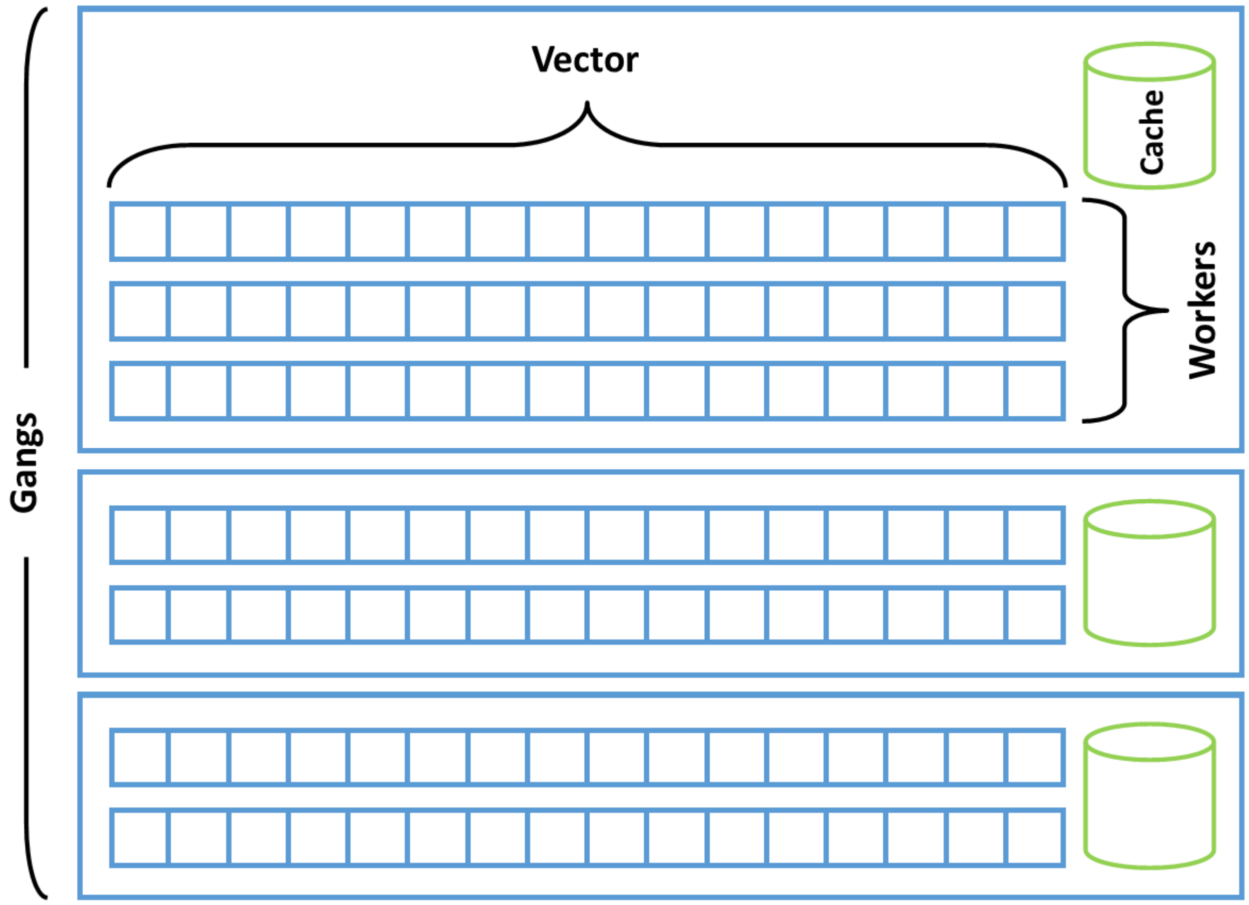


Figure 5.1: OpenACC's Three Levels of Parallelism

Mapping Parallelism to the Hardware

With some understanding of how the underlying accelerator hardware works it's possible to inform that compiler how it should map the loop iterations into parallelism on the hardware. It's worth restating that the more detail the compiler is given about how to map the parallelism onto a particular accelerator the less performance portable the code will be.

As discussed earlier in this guide the `loop` directive is intended to give the compiler additional information about the next loop in the code. In addition to the clauses shown before, which were intended to ensure correctness, the clauses below inform the compiler which level of parallelism should be used to for the given loop.

- Gang clause - partition the loop across gangs
- Worker clause - partition the loop across workers
- Vector clause - vectorize the loop

- Seq clause - do not partition this loop, run it sequentially instead

These directives may also be combined on a particular loop. For example, a `gang vector` loop would be partitioned across gangs, each of which with 1 worker implicitly, and then vectorized. The OpenACC specification enforces that the outermost loop must be a gang loop, the innermost parallel loop must be a vector loop, and a worker loop may appear in between. A sequential loop may appear at any level.

```
#pragma acc parallel loop gang
for ( i=0; i<N; i++)
  #pragma acc loop vector
  for ( j=0; j<M; j++)
    ;
```

–

```
!$acc parallel loop gang
do j=1,M
  !$acc loop vector
  do i=1,N
```

Informing the compiler where to partition the loops is just one part of optimizing the loops. The programmer may additionally tell the compiler the specific number of gangs, workers, or the vector length to use for the loops. This specific mapping is achieved slightly differently when using the `kernels` directive or the `parallel` directive. In the case of the `kernels` directive, the `gang`, `worker`, and `vector` clauses accept an integer parameter that will optionally inform the compiler how to partition that level of parallelism. For example, `vector(128)` informs the compiler to use a vector length of 128 for the loop.

```
#pragma acc kernels
{
#pragma acc loop gang
for ( i=0; i<N; i++)
  #pragma acc loop vector(128)
  for ( j=0; j<M; j++)
    ;
}
```

```
!$acc kernels
!$acc loop gang
do j=1,M
  !$acc loop vector(128)
  do i=1,N

!$acc end kernels
```

When using the `parallel` directive, the information is presented on the `parallel` directive itself, rather than on each individual loop, in the form of the `num_gangs`, `num_workers, andvector_lengthclauses to theparallel`‘ directive.

```
#pragma acc parallel loop gang vector_length(128)
for ( i=0; i<N; i++)
  #pragma acc loop vector
  for ( j=0; j<M; j++)
    ;
```

```
!$acc parallel loop gang vector_length(128)
do j=1,M
  !$acc loop vector(128)
  do i=1,N
```

Since these mappings will vary between different accelerator, the `loop` directive accepts a `device_type` clause, which will inform the compiler that these clauses only apply to a particular device time. Clauses after a `device_type` clause up until either the next `device_type` or the end of the directive will apply only to the specified device. Clauses that appear before all `device_type` clauses are considered default values, which will be used if they are not overridden by a later clause. For example, the code below specifies that a vector length of 128 should be used on devices of type `acc_device_nvidia` or a vector length of 256 should be used on devices of type `acc_device_radeon`. The compiler will choose a default vector length for all other device types.

```
#pragma acc parallel loop gang vector \
            device_type(acc_device_nvidia) vector_length(128) \
            device_type(acc_device_radeon) vector_length(256)
for (i=0; i<N; i++)
{
  y[i] = 2.0f * x[i] + y[i];
}
```

Collapse Clause

When a code contains tightly nested loops it is frequently beneficial to *collapse* these loops into a single loop. Collapsing loops means that two loops of trip counts N and M respectively will be automatically turned into a single loop with a trip count of N times M. By collapsing two or more parallel loops into a single loop the compiler has an increased amount of parallelism to use when mapping the code to the device. On highly parallel architectures, such as GPUs, this can result in improved performance. Additionally, if a loop lacked sufficient parallelism for the hardware by itself, collapsing it with another loop multiplies the available parallelism. This is especially beneficial on vector loops, since some hardware types will require longer vector lengths to achieve high performance than others. The code below demonstrates how to use the collapse directive.

```
! $acc parallel loop gang collapse (2)
do ie = 1 , nelemd
  do q = 1 , qsize
    ! $acc loop vector collapse (3)
    do k = 1 , nlev
      do j = 1 , np
        do i = 1 , np
          qtmp = elem (ie )% state % qdp (i,j,k,q, n0_qdp )
          vs1tmp = vstar (i,j,k ,1, ie) * elem (ie )% metdet (i,j) * qtmp
```

```
10              vs2tmp = vstar (i,j,k ,2, ie) * elem (ie )% metdet (i,j) * qtmp
11              gv(i,j,k ,1) = ( dinv (i,j ,1 ,1 , ie )* vs1tmp + dinv (i,j ,1 ,2, ie )* vs2tmp )
12              gv(i,j,k ,2) = ( dinv (i,j ,2 ,1 , ie )* vs1tmp + dinv (i,j ,2 ,2, ie )* vs2tmp )
13            enddo
14          enddo
15        enddo
16      enddo
17    enddo
```

The above code is an excerpt from a real application where collapsing loops extended the parallelism available to be exploited. On line 1, the two outermost loops are collapsed together to make it possible to generate *gangs* across the iterations of both loops, thus making the total number of gangs `nelemd` x `qsize` rather than just `nelemd`. The collapse at line 4 collapses together 3 small loops to increase the possible *vector length*, as none of the loops iterate for enough trips to create a reasonable vector length on the target accelerator. How much this optimization will speed-up the code will vary according to the application and the target accelerator, but it's not uncommon to see large speed-ups by using collapse on loop nests.

Routine Parallelism

A previous chapter introduced the `routine` directive for calling functions and subroutines from OpenACC parallel regions. In that chapter it was assumed that the routine would be called from each loop iteration, therefore requiring a `routine seq` directive. In some cases, the routine itself may contain parallelism that must be mapped to the device. In these cases, the `routine` directive may have a `gang`, `worker`, or `vector` clause instead of `seq` to inform the compiler that the routine will contain the specified level of parallelism. When the compiler then encounters the call site of the affected routine, it will then know how it can parallelize the code to use the routine.

Case Study - Optimize Loops

This case study will focus on a different algorithm than the previous chapters. When a compiler has sufficient information about loops to make informed decisions, it's frequently difficult to improve the performance of a given parallel loop by more than a few percent. In some cases, the code lacks the information necessary for the compiler to make informed optimization decisions. In these cases, it's often possible for a developer to optimize the parallel loops significantly by informing the compiler how to decompose and distribute the loops to the hardware.

The code used in this section implements a sparse, matrix-vector product (SpMV) operation. This means that a matrix and a vector will be multiplied together, but the matrix has very few elements that are not zero (it is *sparse*), meaning that calculating these values is unnecessary. The matrix is stored in a Compress Sparse Row (CSR) format. In CSR the sparse array, which may contain a significant number of cells whose value is zero, thus wasting a significant amount of memory, is stored using three, smaller arrays: one containing the non-zero values from the matrix, a second that describes where in a given row these non-zero elements would reside, and a third describing the columns in which the data would reside. The code for this exercise is below.

```
#pragma acc parallel loop
for(int i=0;i<num_rows;i++) {
  double sum=0;
  int row_start=row_offsets[i];
  int row_end=row_offsets[i+1];
  #pragma acc loop reduction(+:sum)
  for(int j=row_start;j<row_end;j++) {
    unsigned int Acol=cols[j];
    double Acoef=Acoefs[j];
    double xcoef=xcoefs[Acol];
    sum+=Acoef*xcoef;
  }
  ycoefs[i]=sum;
}
```

```
  !$acc parallel loop
  do i=1,a%num_rows
    tmpsum = 0.0d0
    row_start = arow_offsets(i)
    row_end   = arow_offsets(i+1)-1
    !$acc loop reduction(+:tmpsum)
    do j=row_start,row_end
      acol = acols(j)
      acoef = acoefs(j)
      xcoef = x(acol)
      tmpsum = tmpsum + acoef*xcoef
    enddo
    y(i) = tmpsum
  enddo
```

One important thing to note about this code is that the compiler is unable to determine how many non-zeros each row will contain and use that information in order to schedule the loops. The developer knows, however, that the number of non-zero elements per row is very small and this detail will be key to achieving high performance.

NOTE: Because this case study features optimization techniques, it is necessary to perform optimizations that may be beneficial on one hardware, but not on others. This case study was performed using the PGI 2015 compiler on an NVIDIA Tesla K40 GPU. These same techniques may apply on other architectures, particularly those similar to NVIDIA GPUs, but it will be necessary to make certain optimization decisions based on the particular accelerator in use.

In examining the compiler feedback from the code shown above, I know that the compiler has chosen to use a vector length of 256 on the innermost loop. I could have also obtained this information from a runtime profile of the application.

```
matvec(const matrix &, const vector &, const vector &):
      7, include "matrix_functions.h"
          12, Generating present(row_offsets[:],cols[:],Acoefs[:],xcoefs[:],ycoefs[:])
              Accelerator kernel generated
              15, #pragma acc loop gang /* blockIdx.x */
              20, #pragma acc loop vector(256) /* threadIdx.x */
                  Sum reduction generated for sum
          12, Generating Tesla code
          20, Loop is parallelizable
```

Based on my knowledge of the matrix, I know that this is significantly larger than the typical number of non-zeros per row, so many of the *vector lanes* on the accelerator will be wasted because there's not sufficient work for them. The first thing to try in order to improve performance is to adjust the vector length used on the innermost loop. I happen to know that the compiler I'm using will restrict me to using multiples of the *warp size* (the minimum SIMT execution size on NVIDIA GPUs) of this processor, which is 32. This detail will vary according to the accelerator of choice. Below is the modified code using a vector length of 32.

```c
#pragma acc parallel loop vector_length(32)
for(int i=0;i<num_rows;i++) {
  double sum=0;
  int row_start=row_offsets[i];
  int row_end=row_offsets[i+1];
  #pragma acc loop vector reduction(+:sum)
  for(int j=row_start;j<row_end;j++) {
    unsigned int Acol=cols[j];
    double Acoef=Acoefs[j];
    double xcoef=xcoefs[Acol];
    sum+=Acoef*xcoef;
  }
  ycoefs[i]=sum;
}
```

```fortran
!$acc parallel loop vector_length(32)
do i=1,a%num_rows
  tmpsum = 0.0d0
  row_start = arow_offsets(i)
  row_end   = arow_offsets(i+1)-1
  !$acc loop vector reduction(+:tmpsum)
  do j=row_start,row_end
    acol = acols(j)
    acoef = acoefs(j)
    xcoef = x(acol)
    tmpsum = tmpsum + acoef*xcoef
  enddo
  y(i) = tmpsum
enddo
```

Notice that I have now explicitly informed the compiler that the innermost loop should be a vector loop, to ensure that the compiler will map the parallelism exactly how I wish. I can try different vector lengths to find the optimal value for my accelerator by modifying the `vector_length` clause. Below is a graph showing the relative speed-up of varying the vector length compared to the compiler-selected value.

Notice that the best performance comes from the smallest vector length. Again, this is because the number of non-zeros per row is very small, so a small vector length results in fewer wasted compute resources. On the particular chip I'm using, the smallest possible vector length, 32, achieves the best possible performance. On this particular accelerator, I also know that the hardware will not perform efficiently at this vector length unless we can identify further parallelism another way. In this case, we can use the *worker* level of parallelism to fill each *gang* with more of these short vectors. Below is the modified code.

```c
#pragma acc parallel loop gang worker num_workers(32) vector_length(32)
for(int i=0;i<num_rows;i++) {
  double sum=0;
  int row_start=row_offsets[i];
  int row_end=row_offsets[i+1];
```

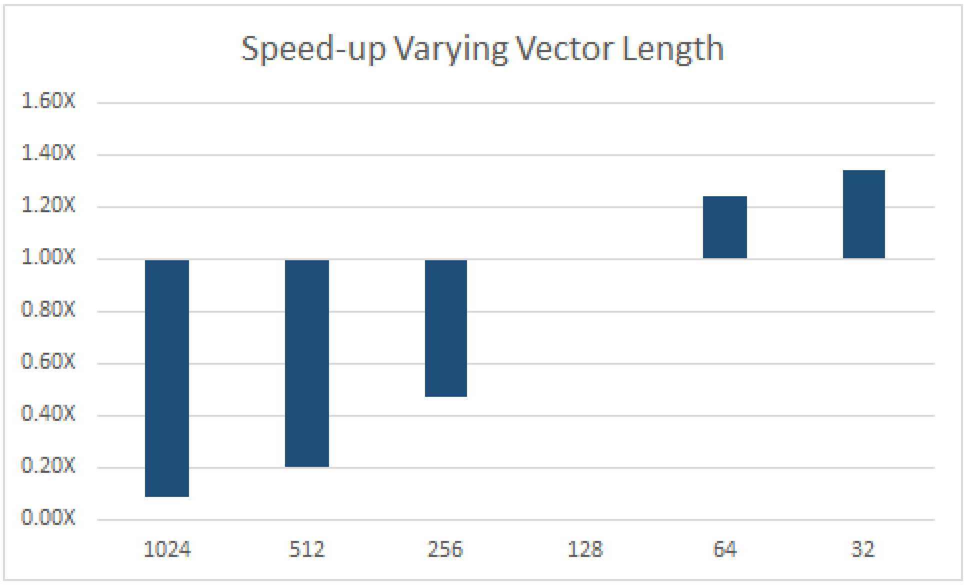


Figure 5.2: Relative speed-up from varying vector_length from the default value of 128

```
    #pragma acc loop vector
    for(int j=row_start;j<row_end;j++) {
      unsigned int Acol=cols[j];
      double Acoef=Acoefs[j];
      double xcoef=xcoefs[Acol];
      sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
  }
```

```
  !$acc parallel loop gang worker num_workers(32) vector_length(32)
  do i=1,a%num_rows
    tmpsum = 0.0d0
    row_start = arow_offsets(i)
    row_end   = arow_offsets(i+1)-1
    !$acc loop vector reduction(+:tmpsum)
    do j=row_start,row_end
      acol = acols(j)
      acoef = acoefs(j)
      xcoef = x(acol)
      tmpsum = tmpsum + acoef*xcoef
    enddo
    y(i) = tmpsum
  enddo
```

In this version of the code, I've explicitly mapped the outermost look to both gang and worker parallelism and will vary the number of workers using the `num_workers` clause. The results follow.

On this particular hardware, the best performance comes from a vector length of 32 and 32 workers. This turns out to be the maximum amount of parallelism that the particular accelerator being used supports

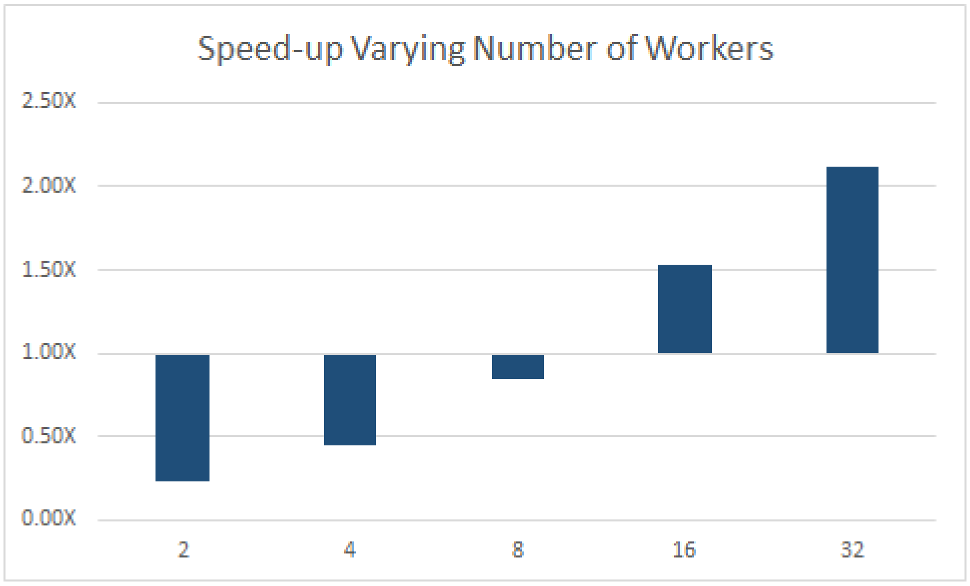


Figure 5.3: Speed-up from varying number of workers for a vector length of 32.

within a gang. In this case, we observed a 1.3X speed-up from decreasing the vector length and another 2.1X speed-up from varying the number of workers within each gang, resulting in an overall 2.9X performance improvement from the untuned OpenACC code.

Best Practice: Although not shown in order to save space, it's generally best to use the `device_type` clause whenever specifying the sorts of optimizations demonstrated in this section, because these clauses will likely differ from accelerator to accelerator. By using the `device_type` clause it's possible to provide this information only on accelerators where the optimizations apply and allow the compiler to make its own decisions on other architectures. The OpenACC specification specifically suggests `nvidia`, `radeon`, and `xeonphi` as three common device type strings.

Chapter 6

OpenACC Interoperability

The authors of OpenACC recognized that it may sometimes be beneficial to mix OpenACC code with code accelerated using other parallel programming languages, such as CUDA or OpenCL, or accelerated math libraries. This interoperability means that a developer can choose the programming paradigm that makes the most sense in the particular situation and leverage code and libraries that may already be available. Developers don't need to decide at the begining of a project between OpenACC *or* something else, they can choose to use OpenACC *and* other technologies.

The Host Data Region

The first method for interoperating between OpenACC and some other code is by managing all data using OpenACC, but calling into a function that requires device data. For the purpose of example the `cublasSaxpy` routine will be used in place of writing a *saxpy* routine, as was shown in an earlier chapter. This routine is freely provided by Nvidia for their hardware in the CUBLAS library. Most other vendors provide their own, tuned library.

The `host_data` region gives the programmer a way to expose the device address of a given array to the host for passing into a function. This data must have already been moved to the device previously. The `host_data` region accepts only the `use_device` clause, which specifies which device variables should be exposed to the host. In the example below, the arrays `x` and `y` are placed on the device via a `data` region and then initialized in an OpenACC loop. These arrays are then passed to the `cublasSaxpy` function as device pointers using the `host_data` region.

```
#pragma acc data create(x[0:n]) copyout(y[0:n])
{
  #pragma acc kernels
  {
    for( i = 0; i < n; i++)
    {
      x[i] = 1.0f;
      y[i] = 0.0f;
    }
  }

  #pragma acc host_data use_device(x,y)
  {
    cublasSaxpy(n, 2.0, x, 1, y, 1);
    }
  }
```

The call to `cublasSaxpy` can be changed to any function that expects device memory as parameter.

Using Device Pointers

Because there is already a large ecosystem of accelerated applications using languages such as CUDA or OpenCL it may also be necessary to add an OpenACC region to an existing accelerated application. In this case the arrays may be managed outside of OpenACC and already exist on the device. In this case OpenACC provides the `deviceptr` data clause, which may be used where any data clause may appear. This clause informs the compiler that the variables specified are already device on the device and no other action needs to be taken on them. The example below uses the `acc_malloc` function, which allocates device memory and returns a pointer, to allocate an array only on the device and then uses that array within an OpenACC region.

```
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
  #pragma acc kernels deviceptr(x,y)
  {
    for(int i=0; i<n; i++)
    {
      y[i] += a*x[i];
    }
  }
}
void set(int n, float val, float * restrict arr)
{
#pragma acc kernels deviceptr(arr)
  {
    for(int i=0; i<n; i++)
    {
      arr[i] = val;
    }
  }
}
int main(int argc, char **argv)
{
  float *x, *y, tmp;
  int n = 1<<20;

  x = acc_malloc((size_t)n*sizeof(float));
  y = acc_malloc((size_t)n*sizeof(float));

  set(n,1.0f,x);
  set(n,0.0f,y);

  saxpy(n, 2.0, x, y);
  acc_memcpy_from_device(&tmp,y,(size_t)sizeof(float));
  printf("%f\n",tmp);
  acc_free(x);
  acc_free(y);
```

```
    return 0;
  }
```

Notice that in the `set` and `saxpy` routines, where the OpenACC compute regions are found, each compute region is informed that the pointers being passed in are already device pointers by using the `deviceptr` keyword. This example also uses the `acc_malloc`, `acc_free`, and `acc_memcpy_from_device` routines for memory management. Although the above example uses `acc_malloc` and `acc_memcpy_from_device`, which are provided by the OpenACC specification for portable memory management, a device-specific API may have also been used, such as `cudaMalloc` and `cudaMemcpy`.

Obtaining Device and Host Pointer Addresses

OpenACC provides the `acc_deviceptr` and `acc_hostptr` function calls for obtaining the device and host addresses of pointers based on the host and device addresses, respectively. These routines require that the addresses actually have corresponding addresses, otherwise they will return NULL.

Additional Vendor-Specific Interoperability Features

The OpenACC specification suggests several features that are specific to individual vendors. While implementations are not required to provide the functionality, it's useful to know that these features exist in some implementations. The purpose of these features are to provide interoperability with the native runtime of each platform. Developers should refer to the OpenACC specification and their compiler's documentation for a full list of supported features.

Asynchronous Queues and CUDA Streams (NVIDIA)

As demonstrated in the next chapter, asynchronous work queues are frequently an important way to deal with the cost of PCIe data transfers on devices with distinct host and device memory. In the NVIDIA CUDA programming model asynchronous operations are programmed using CUDA streams. Since developers may need to interoperate between CUDA streams and OpenACC queues, the specification suggests two routines for mapping CUDA streams and OpenACC asynchronous queues.

The `acc_get_cuda_stream` function accepts an integer async id and returns a CUDA stream object (as a void*) for use as a CUDA stream.

The `acc_set_cuda_stream` function accepts an integer async handle and a CUDA stream object (as a void*) and maps the CUDA stream used by the async handle to the stream provided.

With these two functions it's possible to place both OpenACC operations and CUDA operations into the same underlying CUDA stream so that they will execute in the appropriate order.

CUDA Managed Memory (NVIDIA)

NVIDIA added support for *CUDA Managed Memory*, which provides a single pointer to memory regardless of whether it is accessed from the host or device, in CUDA 6.0. In many ways managed memory is similar to OpenACC memory management, in that only a single reference to the memory is necessary and the runtime will handle the complexities of data movement. The advantage that managed memory sometimes has it that it is better able to handle complex data structures, such as C++ classes or structures containing pointers, since pointer references are valid on both the host and the device. More information about CUDA Managed Memory can be obtained from NVIDIA. To use managed memory within an OpenACC program the developer can simply declare pointers to managed memory as device pointers using the `deviceptr` clause so that the OpenACC runtime will not attempt to create a separate device allocation for the pointers.

Using CUDA Device Kernels (NVIDIA)

The `host_data` directive is useful for passing device memory to host-callable CUDA kernels. In cases where it's necessary to call a device kernel (CUDA `__device__` function) from within an OpenACC parallel region it's possible to use the `acc routine` directive to inform the compiler that the function being called is available on the device. The function declaration must be decorated with the `acc routine` directive and the level of parallelism at which the function may be called. In the example below the function `f1dev` is a sequential function that will be called from each CUDA thread, so it is declared `acc routine seq`.

```
// Function implementation
extern "C" __device__ void
f1dev( float* a, float* b, int i ){
  a[i] = 2.0 * b[i];
}

// Function declaration
#pragma acc routine seq
extern "C" void f1dev( float*, float* int );

// Function call-site
#pragma acc parallel loop present( a[0:n], b[0:n] )
for( int i = 0; i < n; ++i )
{
  // f1dev is a __device__ function build with CUDA
  f1dev( a, b, i );
}
```

Chapter 7

Advanced OpenACC Features

This chapter will discuss OpenACC features and techniques that do not fit neatly into other sections of the guide. These techniques are considered advanced, so readers should feel comfortable with the features discussed in previous chapters before proceeding to this chapter.

Asynchronous Operation

In a previous chapter we discussed the necessity to optimize for data locality to reduce the cost of data transfers on systems where the host and accelerator have physically distinct memories. There will always be some amount of data transfers that simply cannot be optimized away and still produce correct results. After minimizing data transfers, it may be possible to further reduce the performance penalty associated with those transfers by overlapping the copies with other operations on the host, device, or both. This can be achieved with OpenACC using the `async` clause. The `async` clause can be added to `parallel`, `kernels`, and `update` directives to specify that once the associated operation has been sent to the accelerator or runtime for execution the CPU may continue doing other things, rather than waiting for the accelerator operation to complete. This may include enqueing additional accelerator operations or computing other work that is unrelated to the work being performed by the accelerator. The code below demonstrates adding the `async` clause to a `parallel loop` and an `update` directive that follows.

```
#pragma acc parallel loop async
for (int i=0; i<N; i++)
{
  c[i] = a[i] + b[i]
}
#pragma acc update self(c[0:N]) async
```

```
!$acc parallel loop async
do i=1,N
  c(i) = a(i) + b(i)
end do
!$acc update self(c) async
```

In the case above, the host thread will enqueue the parallel region into the *default asynchronous queue*, then execution will return to the host thread so that it can also enqueue the `update`, and finally the CPU thread will continue execution. Eventually, however, the host thread will need the results computed on the accelerator and copied back to the host using the `update`, so it must synchronize with the accelerator to ensure that these operations have finished before attempting to use the data. The `wait` directive instructs the runtime to wait for past asynchronous operations to complete before proceeding. So, the above examples can be extended to include a synchronization before the data being copied by the `update` directive proceeds.

```
#pragma acc parallel loop async
for (int i=0; i<N; i++)
{
  c[i] = a[i] + b[i]
}
#pragma acc update self(c[0:N]) async
#pragma acc wait
```

```
!$acc parallel loop async
do i=1,N
  c(i) = a(i) + b(i)
end do
!$acc update self(c) async
!$acc wait
```

While this is useful, it would be even more useful to expose dependencies into these asynchronous operations and the associated waits such that independent operations could potentially be executed concurrently. Both `async` and `wait` have an optional argument for a non-negative, integer number that specifies a queue number for that operation. All operations placed in the same queue will operate in-order, but operations place in different queues may operate in any order. These work queues are unique per-device, so two devices will have distinct queues with the same number. If a `wait` is encountered without an argument, it will wait on all previously enqueued work on that device. The case study below will demonstrate how to use different work queues to achieve overlapping of computation and data transfers.

In addition to being able to place operations in separate queues, it'd be useful to be able to join these queues together at a point where results from both are needed before proceeding. This can be achieved by adding an `async` clause to an `wait`. This may seem unintuitive, so the code below demonstrates how this is done.

```
#pragma acc parallel loop async(1)
for (int i=0; i<N; i++)
{
  a[i] = i;
}
#pragma acc parallel loop async(2)
for (int i=0; i<N; i++)
{
  b[i] = 2*i;
}
#pragma acc wait(1) async(2)
#pragma acc parallel loop async(2)
for (int i=0; i<N; i++)
{
  c[i] = a[i] + b[i]
}
#pragma acc update self(c[0:N]) async(2)
#pragma acc wait
```

```
    !$acc parallel loop async(1)
    do i=1,N
      a(i) = i
    end do
    !$acc parallel loop async(2)
    do i=1,N
      b(i) = 2.0 * i
    end do
    !$acc wait(1) async(2)
    !$acc parallel loop async(2)
    do i=1,N
      c(i) = a(i) + b(i)
    end do
    !$acc update self(c) async(2)
    !$acc wait
```

The above code initializes the values contained in `a` and `b` using separate work queues so that they may potentially be done independently. The `wait(1) async(2)` ensures that work queue 2 does not proceed until queue 1 has completed. The vector addition is then able to be enqueued to the device because the previous kernels will have completed prior to this point. Lastly the code waits for all previous operations to complete. Using this technique we've expressed the dependencies of our loops to maximize concurrency between regions but still give correct results.

Best Practice: The cost of sending an operation to the accelerator for execution is frequently quite high on offloading accelerators, such as GPUs connected over a PCIe bus to a host CPU. Once the loops and data transfers within a routine have been tested, it is frequently beneficial to make each parallel region and update asynchrounous and then place a `wait` directive after the last accelerator directive. This allows the runtime to enqueue all of the work immediately, which will reduce how often the accelerator and host must synchronize and reduce the cost of launching work onto the accelerator. It is criticial when implementing this optimization that the developer not leave off the `wait` after the last accelerator directive, otherwise the code will be likely to produce incorrect results. This is such a beneficial optimization that some compilers provide a build-time option to enable this for all accelerator directives automatically.

Case Study: Asynchronous Pipelining of a Mandelbrot Set

For this example we will be modifying a simple application that generates a mandelbrot set, such as the picture shown above. Since each pixel of the image can be independently calculated, the code is trivial to parallelize, but because of the large size of the image itself, the data transfer to copy the results back to the host before writing to an image file is costly. Since this data transfer must occur, it'd be nice to overlap it with the computation, but as the code is written below, the entire computation must occur before the copy can occur, therefore there is noting to overlap. *(Note: The `mandelbrot` function is a sequential function used to calculate the value of each pixel. It is left out of this chapter to save space, but is included in the full examples.)*

```
    #pragma acc parallel loop
    for(int y=0;y<HEIGHT;y++) {
      for(int x=0;x<WIDTH;x++) {
        image[y*WIDTH+x]=mandelbrot(x,y);
      }
    }
```

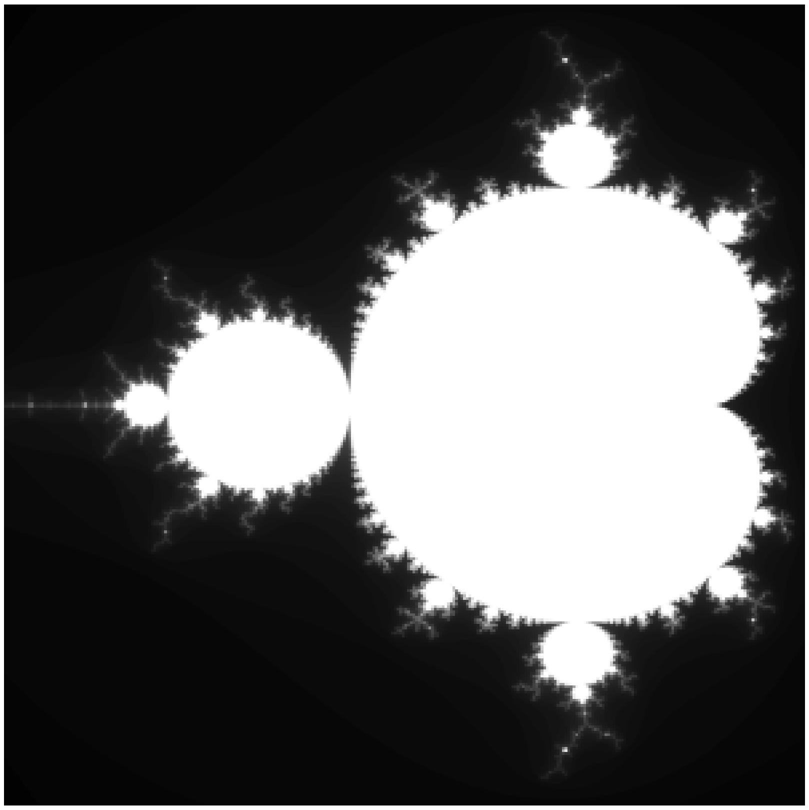


Figure 7.1: Mandelbrot Set Output

```
!$acc parallel loop
do iy=1,width
  do ix=1,HEIGHT
    image(ix,iy) = min(max(int(mandelbrot(ix-1,iy-1)),0),MAXCOLORS)
  enddo
enddo
```

Since each pixel is independent of each other, it's possible to use a technique known as pipelining to break up the generation of the image into smaller parts, which allows the output from each part to be copied while the next part is being computed. The figure below demonstrates an idealized pipeline where the computation and copies are equally sized, but this rarely occurs in real applications. By breaking the operation into two parts, the same amount of data is transferred, but all but the first and last transfers can be overlapped with computation. The number and size of these smaller chunks of work can be adjusted to find the value that provides the best performance.

The mandelbrot code can use this same technique by chunking up the image generation and data transfers into smaller, independent pieces. This will be done in multiple steps to reduce the likelihood of introducing an error. The first step is to introduce a blocking loop to the calculation, but keep the data transfers the same. This will ensure that the work itself is properly divided to give correct results. After each step the developer should build and run the code to ensure the resulting image is still correct.

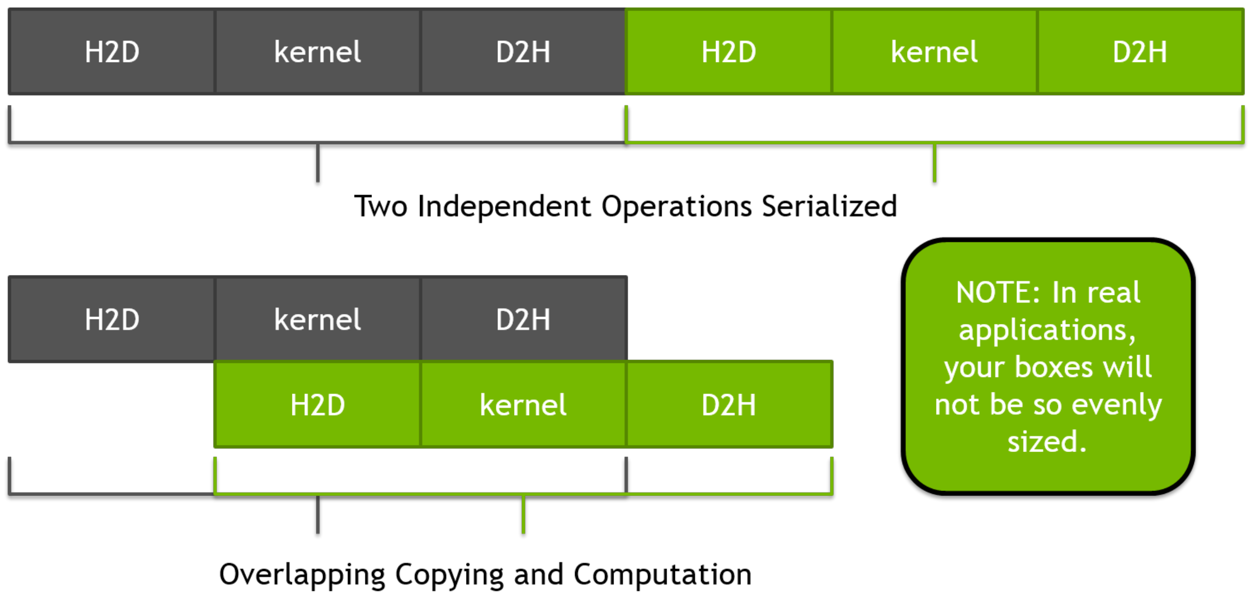

H2D
kernel
D2H
H2D
kernel
D2H
Two Independent Operations Serialized
H2D
kernel
D2H
H2D
kernel
D2H
NOTE: In real applications, your boxes will not be so evenly sized.
Overlapping Copying and Computation

Figure 7.2: Idealized Pipeline Showing Overlapping of 2 Independent Operations

Step 1: Blocking Computation

The first step in pipelining the image generation is to introduce a loop that will break the computation up into chunks of work that can be generated independently. To do this, we will need decide how many blocks of work is desired and use that to determine the starting and ending bounds for each block. Next we introduce an additional loop around the existing two and modify the y loop to only operate within the current block of work by updating its loop bounds with what we've calculated as the starting and ending values for the current block. The modified loop nests are shown below.

```
int num_blocks = 8;
for(int block = 0; block < num_blocks; block++ ) {
  int ystart = block * (HEIGHT/num_blocks),
      yend   = ystart + (HEIGHT/num_blocks);
#pragma acc parallel loop
  for(int y=ystart;y<yend;y++) {
    for(int x=0;x<WIDTH;x++) {
      image[y*WIDTH+x]=mandelbrot(x,y);
    }
  }
}
```

```
num_batches=8
batch_size=WIDTH/num_batches
do yp=0,num_batches-1
  ystart = yp * batch_size + 1
  yend   = ystart + batch_size - 1
  !$acc parallel loop
  do iy=ystart,yend
    do ix=1,HEIGHT
```

```
          image(ix,iy) = min(max(int(mandelbrot(ix-1,iy-1)),0),MAXCOLORS)
        enddo
      enddo
    enddo
```

At this point we have only confirmed that we can successfully generate each block of work independently. The performance of this step should not be noticably better than the original code and may be worse.

Step 2: Blocking Data Transfers

The next step in the process is to break up the data transfers to and from the device in the same way the computation has already been broken up. To do this we will first need to introduce a data region around the blocking loop. This will ensure that the device memory used to hold the image will remain on the device for all blocks of work. Since the initial value of the image array isn't important, we use a `create` data clause to allocate an uninitialized array on the device. Next we use the `update` directive to copy each block of the image from the device to the host after it has been calculated. In order to do this, we need to determine the size of each block to ensure that we update only the part of the image that coincides with the current block of work. The resulting code at the end of this step is below.

```
int num_blocks = 8, block_size = (HEIGHT/num_blocks)*WIDTH;
#pragma acc data create(image[WIDTH*HEIGHT])
for(int block = 0; block < num_blocks; block++ ) {
  int ystart = block * (HEIGHT/num_blocks),
      yend   = ystart + (HEIGHT/num_blocks);
#pragma acc parallel loop
  for(int y=ystart;y<yend;y++) {
    for(int x=0;x<WIDTH;x++) {
      image[y*WIDTH+x]=mandelbrot(x,y);
    }
  }
#pragma acc update self(image[block*block_size:block_size])
}
```

```
num_batches=8
batch_size=WIDTH/num_batches
call cpu_time(startt)
!$acc data create(image)
do yp=0,NUM_BATCHES-1
  ystart = yp * batch_size + 1
  yend   = ystart + batch_size - 1
  !$acc parallel loop
  do iy=ystart,yend
    do ix=1,HEIGHT
      image(ix,iy) = mandelbrot(ix-1,iy-1)
    enddo
  enddo
  !$acc update self(image(:,ystart:yend))
enddo
!$acc end data
```

By the end of this step we are calculating and copying each block of the image independently, but this is still being done sequentially, each block after the previous. The performance at the end of this step is generally comparable to the original version.

Step 3: Overlapping Computation and Transfers

The last step of this case study is to make the device operations asynchronous so that the independent copies and computation can happen simultaneously. To do this we will use asynchronous work queues to ensure that the computation and data transfer within a single block are in the same queue, but separate blocks land in different queues. The block number is a convenient asynchronous handle to use for this change. Of course, since we're now operating completely asynchronously, it's critical that we add a `wait` directive after the block loop to ensure that all work completes before we attempt to use the image data from the host. The modified code is found below.

```c
int num_blocks = 8, block_size = (HEIGHT/num_blocks)*WIDTH;
#pragma acc data create(image[WIDTH*HEIGHT])
for(int block = 0; block < num_blocks; block++ ) {
  int ystart = block * (HEIGHT/num_blocks),
      yend   = ystart + (HEIGHT/num_blocks);
#pragma acc parallel loop async(block)
  for(int y=ystart;y<yend;y++) {
    for(int x=0;x<WIDTH;x++) {
      image[y*WIDTH+x]=mandelbrot(x,y);
    }
  }
#pragma acc update self(image[block*block_size:block_size]) async(block)
}
#pragma acc wait
```

```fortran
num_batches=8
batch_size=WIDTH/num_batches
call cpu_time(startt)
!$acc data create(image)
do yp=0,NUM_BATCHES-1
  ystart = yp * batch_size + 1
  yend   = ystart + batch_size - 1
  !$acc parallel loop async(yp)
  do iy=ystart,yend
    do ix=1,HEIGHT
      image(ix,iy) = mandelbrot(ix-1,iy-1)
    enddo
  enddo
  !$acc update self(image(:,ystart:yend)) async(yp)
enddo
!$acc wait
!$acc end data
```

With this modification it's now possible for the computational part of one block to operate simultaneously as the data transfer of another. The developer should now experiment with varying block sizes to determine what the optimal value is on the architecture of interest. It's important to note, however, that on some architectures the cost of creating an asynchronous queue the first time its used can be quite expensive. In long-running applications, where the queues may be created once at the beginning of a many-hour run and reused throughout, this cost is amortized. In short-running codes, such as the demonstration code used in this chapter, this cost may outweigh the benefit of the pipelining. Two solutions to this are to introduce a simple block loop at the beginning of the code that pre-creates the asynchronous queues before the timed section, or to use a modulus operation to reuse the same smaller number of queues among all of the blocks. For instance, by using the block number modulus 2 as the asynchronous handle, only two queues will be used and the cost of creating those queues will be amortized by their reuse. Two queues is generally sufficient to see a gain in performance, since it still allows computation and updates to overlap, but the developer should experiment to find the best value on a given machine.

Below we see a screenshot showing before and after profiles from applying these changes to the code on an NVIDIA GPU platform. Similar results should be possible on any acclerated platform. Using 64 blocks and two asynchronous queues, as shown below, roughly a 2X performance improvement was observed on the test machine over the performance without pipelining.

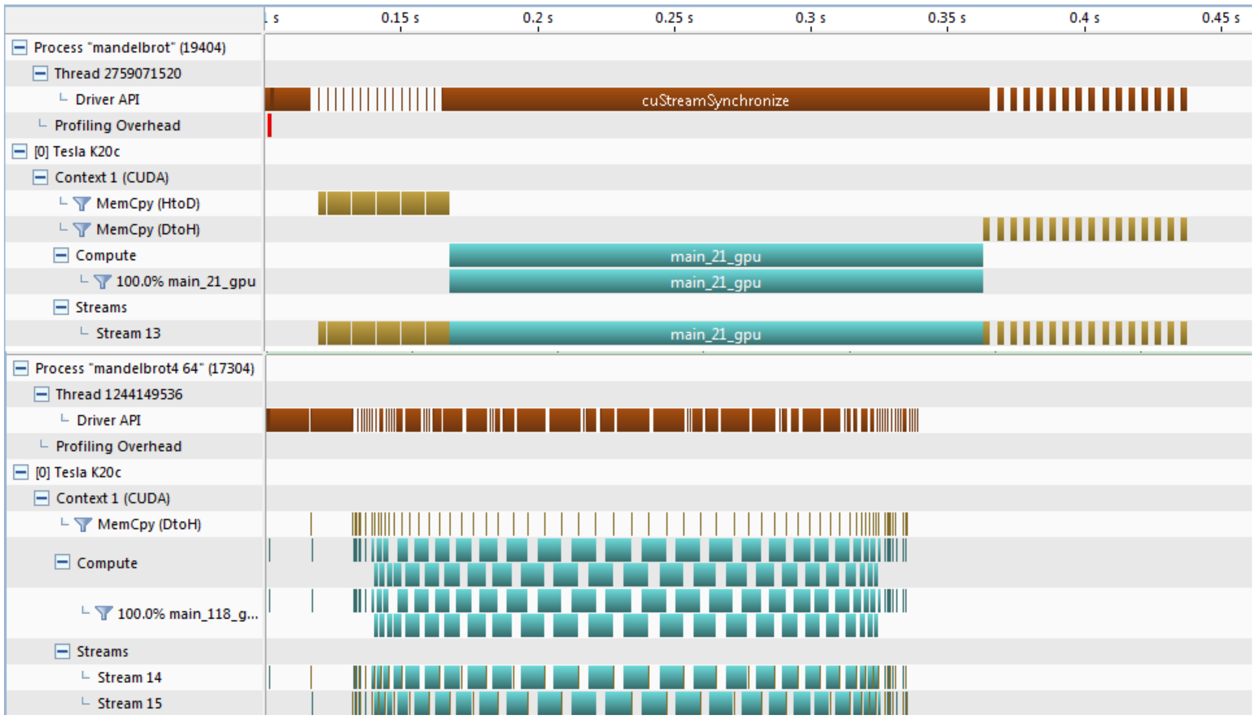


Figure 7.3: Visual profiler timelines for the original mandelbrot code (Top) and the pipelined code using 64 blocks over 2 asynchronous queues (Bottom).

Multi-device Programming

For systems containing more than accelerator, OpenACC provides and API to make operations happen on a particular device. In case a system contains accelerators of different types, the specification also allows for querying and selecting devices of a specific architecture.

acc_get_num_devices()

The `acc_get_num_devices()` routine may be used to query how many devices of a given architecture are available on the system. It accepts one parameter of type `acc_device_t` and returns a integer number of devices.

acc_get_device_num() and acc_set_device_num()

The `acc_get_device_num()` routines query the current device that will be used of a given type and returns the integer identifier of that device. The `acc_set_device_num()` accepts two parameters, the desired device number and device type. Once a device number has been set, all operations will be sent to the specified device until a different device is specified by a later call to `acc_set_device_num()`.

acc_get_device_type() and acc_set_device_type()

The `acc_get_device_type()` routine takes no parameters and returns the device type of the current default device. The `acc_set_device_type()` specifies to the runtime the type of device that the runtime should use for accelerator operations, but allows the runtime to choose which device of that type to use.

Multi-device Programming Example

As a example of multi-device programming, it's possible to further extend the mandelbrot example used previously to send different blocks of work to different accelerators. In order to make this work, it's necessary to ensure that device copies of the data are created on each device. We will do this by replacing the structured `data` region in the code with an unstructured `enter data` directive for eac device, using the `acc_set_device_num()` function to specify the device for each `enter data`. For simplicity, we will allocate the full image array on each device, although only a part of the array is actually needed. When the memory requirements of the application is large, it will be necessary to allocate just the pertinent parts of the data on each accelerator.

Once the data has been created on each device, a call to `acc_get_device_type()` in the blocking loop, using a simple modulus operation to select which device should receive each block, will sent blocks to different devices.

Lastly it's necessary to introduce a loop over devices to wait on each device to complete. Since the `wait` directive is per-device, the loop will once again use `acc_get_device_type()` to select a device to wait on, and then use an `exit data` directive to deallocate the device memory. The final code is below.

```
// Allocate arrays on both devices
for (int gpu=0; gpu < 2 ; gpu ++)
{
  acc_set_device_num(gpu,acc_device_nvidia);
#pragma acc enter data create(image[:bytes])
}

// Distribute blocks between devices
for(int block=0; block < numblocks; block++)
{
  int ystart = block * blocksize;
  int yend   = ystart + blocksize;
  acc_set_device_num(block%2,acc_device_nvidia);
#pragma acc parallel loop async(block)
  for(int y=ystart;y<yend;y++) {
    for(int x=0;x<WIDTH;x++) {
      image[y*WIDTH+x]=mandelbrot(x,y);
    }
  }
#pragma acc update self(image[ystart*WIDTH:WIDTH*blocksize]) async(block)
}

// Wait on each device to complete and then deallocate arrays
for (int gpu=0; gpu < 2 ; gpu ++)
{
  acc_set_device_num(gpu,acc_device_nvidia);
#pragma acc wait
#pragma acc exit data delete(image)
}
```

```
      batch_size=WIDTH/num_batches
      do gpu=0,1
        call acc_set_device_num(gpu,acc_device_nvidia)
        !$acc enter data create(image)
      enddo
      do yp=0,NUM_BATCHES-1
        call acc_set_device_num(mod(yp,2),acc_device_nvidia)
        ystart = yp * batch_size + 1
        yend   = ystart + batch_size - 1
        !$acc parallel loop async(yp)
        do iy=ystart,yend
          do ix=1,HEIGHT
            image(ix,iy) = mandelbrot(ix-1,iy-1)
          enddo
        enddo
        !$acc update self(image(:,ystart:yend)) async(yp)
      enddo
      do gpu=0,1
        call acc_set_device_num(gpu,acc_device_nvidia)
        !$acc wait
        !$acc exit data delete(image)
      enddo
```

Although this example over-allocates device memory by placing the entire image array on the device, it does serve as a simple example of how the `acc_set_device_num()` routine can be used to operate on a machine with multiple devices. In production codes the developer will likely want to partition the work such that only the parts of the array needed by a specific device are available there. Additionally, by using CPU threads it may be possible to issue work to the devices more quickly and improve overall performance. Figure 7.3 shows a screenshot of the NVIDIA Visual Profiler showing the mandelbrot computation divided across two NVIDIA GPUs.

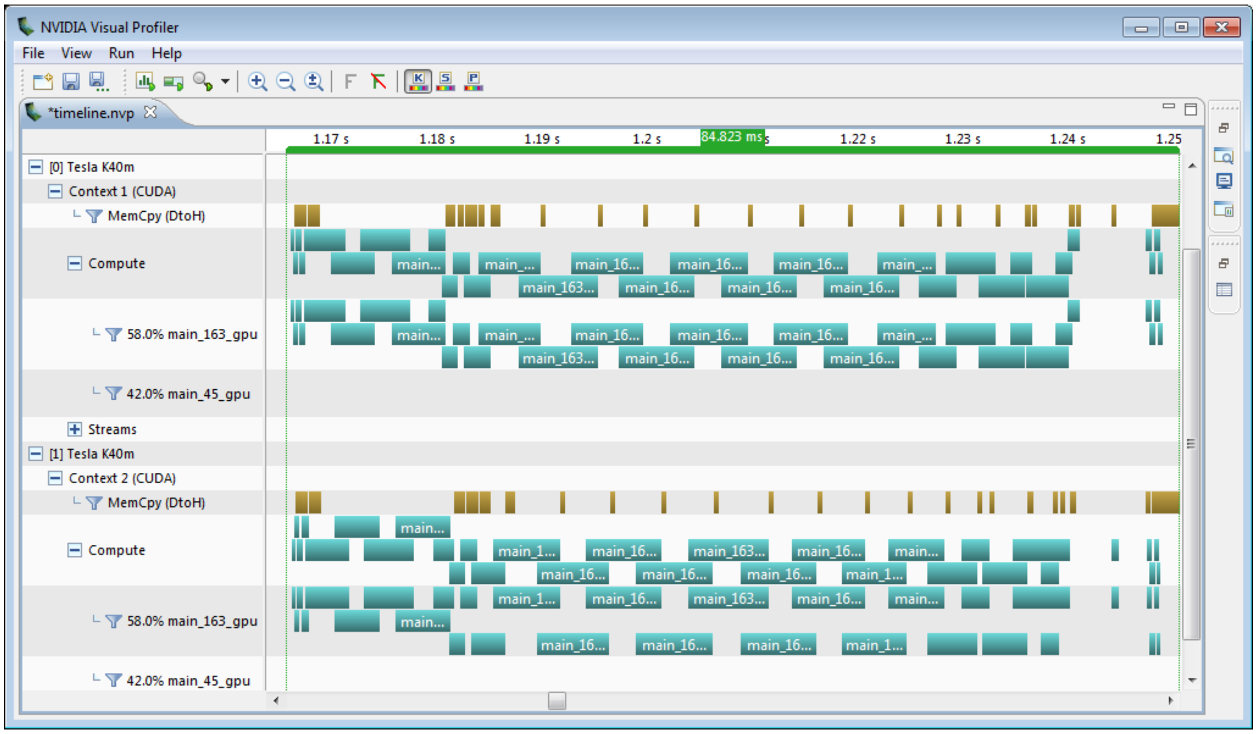


Figure 7.4: NVIDIA Visual Profiler timeline for multi-device mandelbrot

Appendix A

References

- OpenACC.org
- OpenACC on the NVIDIA Parallel Forall Blog
- PGI Insider Newsletter
- OpenACC at the NVIDIA GPU Technology Conference
- OpenACC on Stack Exchange