Portable Parallel Programming with OpenACC

Team Kent Ridge

Learning Outcomes

1. Exploit multiple levels of parallelization with OpenACC compute constructs.
2. Optimize distributed memory movements with data constructs.
3. Optimize a 2D Laplace Jacobi iterator for a H-100 GPU.

Using the SoC Compute Cluster

We will be using **NUS School of Computing's Compute Cluster** for their discrete NVIDIA GPUs. Note that we do not directly control these machines - they are a resource allocated through Slurm, made available to the *whole* School for research purposes and centrally managed by SoC IT.

We have also pre-installed the Nvidia HPC SDK (NVHPC) via Spack on the `tkr` account on SoCC. Before we begin, please do the following:

1. SSH into `tkr@xlogin.comp.nus.edu.sg`
2. Run `spack load nvhpc` to set environment variables (adding commands and tools to `$PATH`.)
3. Check that the NVC++ compiler works: `nvc++ --version`
4. Copy and unzip the `openacc_tut.tar.gz` archive in a new directory. This will be your private workspace for this tutorial.

You are welcome to attempt this tutorial in a different environment, in which case the requirements are a Nvidia CUDA-capable GPU and the Nvidia HPC SDK. You will optionally need a multicore CPU host and `hwloc`.

Please now access our documentation on how to access the GPUs on the SoC Compute Cluster:
`https://comp.nus.edu.sg/~cs3210/student-guide/soc-gpus/`.

Additional information about the cluster can be found here:
`https://dochub.comp.nus.edu.sg/cf/guides/compute-cluster/start`.

Part 1: Introduction to OpenACC

Ye Olde Matrix Multiplication

Students of CS3210 may be familiar with the matrix multiplication example, particularly in the context of OpenMP parallelization (`mm-omp.cpp`).

```
void mm(matrix a, matrix b, matrix result)
{
    int i, j, k;
    // Parallelize the multiplication
    // Iterations of the outer-most loop are divided
    //   amongst the threads
    // Variables (a, b, result) are shared between threads
    // Variables (i, j, k) are private per-thread
    #pragma omp parallel for \
        shared(a, b, result) private (i, j, k)
    for (i = 0; i < size; i++)
         for (j = 0; j < size; j++)
              for (k = 0; k < size; k++)
                  result[i][j] += a[i][k] * b[k][j];
}
```

- Compile the code in a terminal:

  ```
  $ g++ -fopenmp -Ofast -march=znver3 -o mm-omp mm-omp.cpp
  ```

- Let's run our program on all 128 cores of an AMD EPYC node given a 4096^2-element matrix, too large to be run serially in any reasonable time.

  ```
  $ srun -C xcnf -c 128 ./mm-omp 4096
  ```

The gist of this is that we decompose the outer loop of the matrix computation into thread-mapped **tasks**, each of which concurrently **reads a row from matrix** `a` and **all of matrix** `b` to calculate **a row in the** `result` **matrix**. There is no need for synchronization (no computation in any row depends on the result of another row) or explicit communication (the matrices on a multicore platform are in a shared address space).

How would we do this in **OpenACC**? It's deceptively simple: just swap out the directive! (`mm-acc.cpp`)

```
void mm(matrix a, matrix b, matrix result)
{
    int i, j, k;
    // #pragma omp parallel for \
    //      shared(a, b, result) private (i, j, k)
    #pragma acc parallel loop private(i,j,k)
    for (i = 0; i < size; i++)
        for (j = 0; j < size; j++)
            for (k = 0; k < size; k++)
                result[i][j] += a[i][k] * b[k][j];
}
```

OpenACC is a high-level, directive-based programming model for C/C++ and Fortran, similar to OpenMP. An OpenACC directive begins with `#pragma acc ...` and can contain clauses such as `kernel`, `data copy` and `loop vector`. Directives are applied to the **immediately following statement, loop, or structured code block (scope block)**. Together, they form an **OpenACC construct**.

Let's compile the example and have the compiler parallelize our code.

- Compile the code with the NVC++ compiler:

  ```
  $ nvc++ -acc=multicore -Minfo=acc -fast -march=znver3 \
  $ -o mm-acc-cpu mm-acc.cpp
  ```

  The flag `-acc=multicore` indicates we wish to parallelize for a multicore CPU. `-Minfo=acc` makes the compiler give feedback on what got exactly got parallelized, and how.

- Let's run our parallel version on the AMD EPYC node again.

  ```
  $ srun -C xcnf -c 128 ./mm-acc-cpu 4096
  ```

Compiler Support
Note that the `nvcc` compiler packaged with the CUDA toolkit does not support OpenACC, nor do GCC compilers and runtimes built without offloading features.

Currently, OpenACC production compilers can target hardware platforms including traditional X86 multicore platforms; accelerators such as Nvidia and AMD GPUs; and Advanced RISC Machines (ARM) processors. You can even target systems with heterogeneous nodes composed of a mixture of processing element types, or use it in code alongside OpenMP directives and MPI routines.

So what do we need to modify in our code such that it can run on, say, a 30,000USD Nvidia H100 accelerator? Surprisingly, *nothing*!

- Check out the CUDA environment, GPU hardware specifications and driver information on the H100 node.

```
$ srun -G h100-96 bash -c "nvidia-smi; nvaccelinfo"
```

- Recompile the code with the NVC++ compiler:

```
$ srun -G h100-96 -- nvc++ -acc=gpu -Minfo=acc -fast
$ -o mm-acc-gpu mm-acc.cpp
```

  Note the different target under `-acc`.

- Running on our GPU target:

```
$ srun -G h100-96 ./mm-acc-gpu 4096
```

Exercise 1

Compare the execution time with the multi-core run. Which is faster?

Exercise 2

Try profiling the GPU target, e.g. `srun -G h100-96 -- nsys nvprof ./mm3 4096`. Where is the bottleneck?

OpenACC Fundamentals

Before we go on, note that you are free to read more about OpenACC in the best practices guide, the API guide and the official textbook.

Compute Constructs

An OpenACC program consists of several parallel regions, called **compute constructs**. In each parallel block, the compiler may emit code that offloads the computation to a **device**. Outside of these blocks, execution occurs on a host platform, which is assumed to be where the **data** resides or is initialized. Note that the device and host need not be distinct, which is what allowed us to run our matrix multiplication code on multiple cores of the 'host' machine with `"-acc=multicore"`.

Below is an OpenACC Hello World program (`hello-acc.cpp`) showing some of the **runtime library APIs**. You will need to explicitly `#include "openacc.h"` to be able to use them; the directives themselves do not depend on this header.

```
#ifdef _OPENACC
#include <openacc.h>
#endif

int main(int argc, char** argv)
{
    auto type = acc_get_device_type();
    std::string type_s;
    switch (type)
    {
    case acc_device_nvidia:
        type_s = "an Nvidia GPU"; break;
    case acc_device_host:
        type_s = "the host CPU node"; break;
    default:
        sprintf(type_s.data(), "an Unknown device (%d)",
            type);
    }
    printf("Hello world!\nDevice is %s, idx %d of %d\n",
        type_s.c_str(), acc_get_device_num(type),
        acc_get_num_devices(type));

    double v[N];

    #pragma acc parallel
    {
        #pragma acc loop
        for(int i=0; i<N/2; i++) v[i] = sin(v[i]);

        #pragma acc loop
        for(int i=N/2; i<N; i++) v[i] = cos(v[i]);
    }
    return 0;
}
```

Exercise 3

Read the contents of `hello-acc.cpp`. Then, compile and run the `hello-acc.cpp` program as in the previous exercise; on the AMD EPYC node and then on the Nvidia H100. Is there any difference in their output?

Note that here we have **multiple loops** inside a single `parallel` region, in which case they can **both execute concurrently** with no guarantee that the first loop will have finished before the second one starts. This pattern is not guaranteed to be correct if there are inter-dependent computations in each loop (e.g. a write to an element in one loop will race with a read on the same element in another loop). The parallel constructs themselves, however, do have an implicit barrier between them, such that all parallel blocks will run sequentially, one after the other.

Mapping to Three Levels of Parallelism

Because parallel hardware has different places where parallelism can occur - across processors, cores, hyper-threads, NUMA nodes, vector units, blocks and so on — OpenACC provides functionality for you to specify `how` exactly to distribute work across different hardware units. This functionality is implemented in the `gang`, `worker`, and `vector` mapping clauses.

Gangs can best be thought about as groups of workers, wherein different groups/gangs can (only) work independently. Gangs do not allow for synchronization with other gangs and are assumed to not have a backing-shared memory.

Workers are 'threads' within a gang which have shared memory and can synchronize with other workers in the same gang (pause and wait for all workers or a subset of workers to be in the same place in the code, or atomically update to a shared variable between the gang workers).

Vector parallelism has the finest granularity, with an individual instruction operating simultaneously on multiple pieces of data (much like SIMD parallelism on a modern CPU, or warps on NVIDA GPUs). These operations are performed within each thread with reference to a vector length, which specifies how many units of data will be operated on for a single given instruction.

When applied to the `loop` directive, iterations of that loop will be mapped to the hardware corresponding to the clause's level of parallelization. These clauses take an integer value that specifies, respectively, the number of gangs to be used for each kernel or gang loop in the region, the number of workers per gang, and the vector length to use in vector operations. Example:

```
#pragma acc parallel loop num_workers(32) vector_length(128)
for (int f=0; f<A; f++){
    for (int g=0; g<B; g++){
        for (int h=0; h<X; h++){
            work();
        }
    }
}
```

You do not have to specify mapping to hardware via these clauses in order to write a correct OpenACC program. Therefore, it is good practice not to include them until your developed or ported program has been validated and you are ready to start optimizing it for a given hardware architecture.

Table 1 provides a summary of the possible mappings of the gang, worker, and vector to various hardware platforms.

PLATFORM	MAPPING OF ...		
	GANG	WORKER	VECTOR
Multicore CPU	Whole CPU (NUMA domain)	Core	SIMD vector
Manycore CPU (Xeon Phi)	NUMA domain (whole chip or quadrant)	Core	SIMD vector
NVIDIA GPU (CUDA)	Thread block	Warp	Thread
AMD GPU (OpenCL)	Workgroup	Wavefront	Thread

Table 1: Possible mapping of the gang, worker, and vector to various hardware platforms

Note that the mapping actually used is up to the compiler and its runtime environment, and this table is not guaranteed for an actual OpenACC implementation. Though mapping clauses may provide optimized performance for specific hardware, it may come at the expense of how well the application's performance stands up when the application is ported (often termed as its **performance portability**) to other hardware.

Exercise 4

1. Run `srun -C xcnf -c 128 -- bash -c "lscpu; spack load hwloc%aocc && lstopo --no-io --of ascii"`. What does the output tell you? What do you notice about the cache layout?
2. Compile and run `ln-acc.cpp` on the AMD EPYC node. Can you tell what the program is doing? Tune the constant macro `G` to try and improve the runtime performance.

Other Loop Constructs

We'll briefly go through some other loop clauses in OpenACC.

- **Loop Collapse**: When you manually specify how loop iterations should be mapped to the available parallel hardware, the loops you are parallelizing **must have enough iterations to map to the various hardware levels**. You must also specify loop clauses on all loops inside a parallel region.

  As an alternative, you can treat a whole nest of loops as a single loop space to be distributed, specify the parallelism at a single loop level, and allow the compiler to sort out the mapping of those loops to specific hardware.

  The `collapse` keyword for the loop directive, like in OpenMP, takes a single positive number as an argument and uses this number to determine how many loops within the loop nest to collapse into a single loop:

```
#pragma acc parallel loop collapse(3)
for(int i=0; i<8; i++){
    for(int j=0; j<8; j++){
        for(int k=0; k<8; k++){
. . .
        }
    }
}
```

Exercise 5

1. Revisit `mm-acc.cpp`. Recompile it for the GPU. What do you notice in the compiler feedback about how it parallelized the nested loop?
2. Add a `collapse` clause with a value of 2 to the loop directive, like so:

   ```
   #pragma acc parallel loop private(i,j,k) collapse(2)
   ```

   What do you observe about the compiler debug info and execution time now?

- **Routines**: In our examples so far, we have only called functions from outside of compute constructs. Making calls inside compute constructs is a complicated process as the function may need to be compiled in a way to support any arbitrary level of parallelism. For instance, the compiler may need to generate a device version of the function before compute constructs can call the procedure.

  You may use the `routine` directive is to mark a function as potentially parallel. The following code snippet shows three **separate** ways that a function `foo` can be modified with the `routine` directive:

```
// 1. on function declaration
#pragma acc routine
extern void foo(int *a, int *b, int n);

// 2. declare an already prototyped function by name
#pragma acc routine(foo)

// 3. on function definition
#pragma acc routine
void foo(int *a, int *b, int n) {
    #pragma acc loop gang
    for(int j = 0; j < n; ++j)
        a[j] += b[j];
}
```

- **Sequential**: At times a loop may appear parallelizable but contains some **data dependency** that the compiler may not recognize. Because the parallel directive instructs the compiler to parallelize all the code contained within the parallel region, this may result in race conditions.

Additionally, there may be **more nested loop levels than can be supported** by OpenACC with its three levels of parallelism. In such a case, the compiler's choice of which loops to decompose and which to preserve may be suboptimal.

Therefore, OpenACC provides a clause, seq, that you can add to a loop to prevent the compiler from parallelizing or vectorizing that loop, or to control where parallelism happens in the loop nest. You can add seq to any loop directive, but it is **mutually exclusive with `gang`, `worker`, and `vector` clauses** (because they specify parallelism, and seq disables parallelism).

```
#pragma acc parallel loop gang
for (f=0; f<A; f++){
    #pragma acc loop worker
    for (g=0; g<B; g++){
        #pragma acc loop seq
        for (h=0; h<X; h++){
            #pragma acc loop vector
            for (i=0; i<Q; i++)
                ...
```

Exercise 6
Revisit `mm-acc.cpp` once more. Change the construct value to collapse all three loops, i.e. `collapse(3)`. What do you observe now?

- **Reduction**: A number of algorithmic features can cause compilers to not be able to automatically parallelize loops. A common case is when a variable, like a counter or accumulator, is **updated during each loop iteration**. Each iteration would thus have a **dependency to the previous iteration**.

  The `reduction` clause, which can be added to `parallel`, or `loop` directives, specifies that a variable will be **operated on locally for each thread** that is working on it. There will then be a final, typically sub-linear time operation to **accumulate together all the local versions** of the variable. In this way, the compiler parallelizes as much as possible while preserving correctness.

```
total = 0;
#pragma acc parallel loop reduction(+:total)
for(i=0; i<100; i++)
    total += data[i];
```

OpenACC reductions currently only support certain operators on scalar variables, as shown in Table 2. (Reductions are currently not allowed for arrays, array elements, class or structure members.)

C AND C++			
+	*	max	min
&	|	&&	||

Table 2: Supported reduction operations

Exercise 7
The program spmv-acc.cpp is missing someting. Can you spot it? Fix the error, then compile and run it on a GPU.

There are additional options available for the compute directives we've gone though so far; you can read about them in the OpenACC Quick Reference.

Data Constructs

OpenACC is designed to handle environments where the compute constructs are executed on a device that has a **disjoint memory space** from the main program and therefore requires explicit message-passing (**data movement**) between the host and the device.

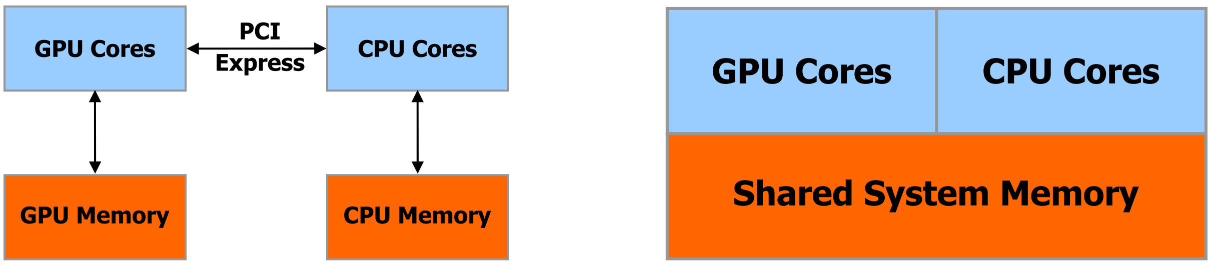

Figure 1: Example memory organization in discrete GPUs (left) and integrated accelerators (right). Data management directives are necessary for OpenACC programs to be portable to discrete platforms.

OpenACC guarantees that the loop control variables (e.g. the index variable) and variables declared inside the loop construct are **predetermined to be** `private`, meaning that one copy per execution element will be created. Scalar variables defined outside of parallel regions and are accessed within them are, by default, considered to be either `private` or `firstprivate` (a private variable that is initialized with the value from before the construct).

For all other cases, although compilers are usually intelligent enough to trace down dependent data and **automatically insert procedure calls to place them on the device for you**, they are quite often fallible in cases such as with **dynamically-allocated indeterminate-size data**.

You may already have, in previous examples, encountered some of the directives and clauses provided by OpenACC to control memory placement. Let's go through them in detail.

There are essentially two types of data directives: **structured** and **unstructured**.

Structured Data Regions

A structured data region defines in a **single lexical scope** when the lifetime/availability of a data object on the **device** (hereafter simply referred to as the **data lifetime**) both begins and ends.

```
#pragma acc data copy(a)
{
  <use a>
}
```

The data lifetime for a begins at the opening curly brace and ends at the closing curly brace. Clauses that follow this syntax are:

- **Present:** The foundation of all the clauses that allocate data is the `present` clause. It is the first part of all data clauses, but it is more of a bookkeeping clause than a data clause: it merely *asserts* that an object is **present on the device's memory** and throws an exception if it is not. This may be useful by itself to ensure that required objects are already on the device so that a computation - one that would normally not be beneficial if data motion were required — can be sent to the device.
- **Create:** The `create` clause creates a data lifetime for the objects listed in the clause on the device. It essentially begins with a `present` clause: if the object does **not** already have memory assigned to it, then the runtime will **allocate memory for the object** and update the *reference count* - the number of data constructs associated with the object. At the end of the lexical scope, the object's memory is freed if its reference count is decremented to zero.
- **Copy:** Arguably the most complex and useful clause, copy begins (and ends) like a `create` clause: ensuring an object has memory allocated for it on the device for the scope of the construct. Then, the copy clause **copies the data from the host to the device** at the beginning of the region. When the region ends, if an object's reference count is about to go to zero, then the data is **copied back to the host** and the memory is freed.

Unstructured Data Regions

An unstructured data region is delimited with a **begin-end pair** (`enter data` and `exit data`) that need not be associated to some lexical scope.

```
void foo(int *array, int n) {
#pragma acc enter data copyin(array[0:n])
}
void bar(int *array, int n) {
#pragma acc exit data copyout(array[0:n])
}
```

The three unstructured clauses are all related to copy:

- **Copy In:** The `copyin` clause does everything the `copy` clause does on the way into a data region, but at the end of the region the data is **not** copied back to the host.
- **Copy Out:** The `copyout` clause does everything the `copy` clause does, except `copyout` does **not** copy the data to the device on entry to the data region.
- **Delete:** The most dangerous clause, `delete` does two things. First it determines whether the object is present; if it is, it erases the object from the device's data environment by forcing the reference count to zero and **releasing the memory**.

Update

There is one more special data directive that can only be called from **within** a data region (unstructured regions included).

For instance, you may be part way through a series of computations on the device and wish to **send modified data back to the host**, to dispatch some intermediate output. The `update` directive copies data between disjoin memory spaces without interfering with the object's memory or lifetime. Supplying the `host` clause copies the listed data from the device to the host thread, and vice versa for the `device` clause.

```
#pragma acc data copy(a)
{
  #pragma acc loop
  ...
  #pragma acc update host(a)
  ...
}
```

Partial Data Transfers

There are two cases when a programmer has to be more specific with data clauses: when the array is **dynamically allocated** or when the programmer only wants to transfer **a part of the array**. (Note: The compiler cannot always determine the size of arrays.) In the first case when the user only declares a pointer and then uses a malloc to attach memory to it, the compiler may not be able to determine the size of the array at compile time if it is not accessed directly with the loop index and thus cannot generate a data transfer of a given size. In the latter case the programmer wants to override the conservative compiler approach for a computation that only operates on part of an array, manually selecting which part(s) of the array to copy.

OpenACC offers the capability for partial data transfers on all data constructs and clauses, which is often termed **array shaping**:

```
#pragma acc data copy(a[0:n])
```

Exercise 8

Study the `spmv` function in `spmv-acc.cpp` again. Try removing data objects from the `copyin` clause, as much as it still results in a correct execution once compiled.

Part 2: A Case Study

In this section, we will make full use of what we've learned so far to try to **parallelize real code**, from the ground up.

Let's look at a real, physical, problem. You have a **square metal plate**. It is initially at zero degrees. This is termed, unsurprisingly, the *initial conditions*. You will **heat two of the edges** in an interesting pattern where you heat the **lower-right corner** (as pictured in Figure 2A) to 100 degrees. You control the two heating elements that lead from this corner such that they go steadily to zero degrees at their farthest edge. The other two edges you will hold at zero degrees. These four edges constitute the *boundary conditions*.

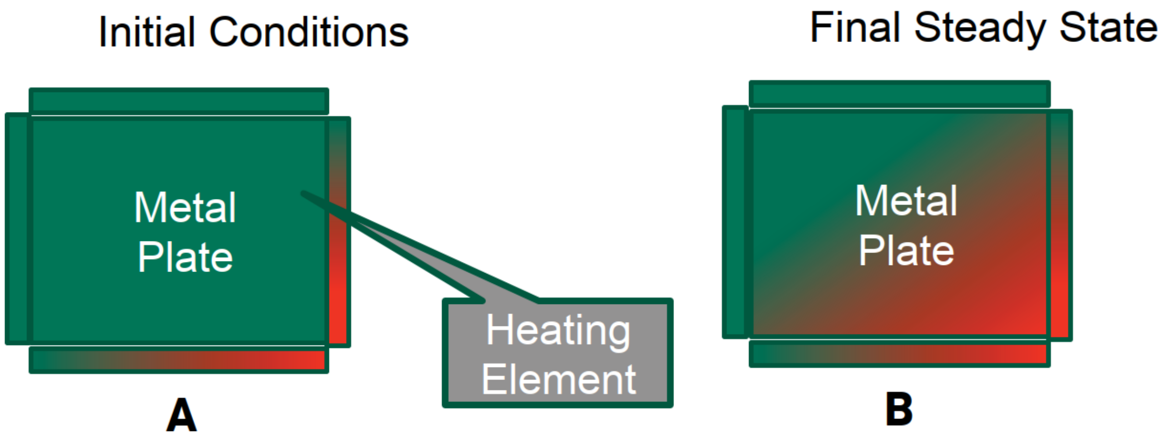

Figure 2: Enter Caption

You would probably guess the numerical solution (the **steady state**) should look something like 2B: very hot corner, a very cold corner, and some kind of gradient in between. The equation that governs heat distribution is the Laplace equation:

$$\nabla^2 T = 0$$

It's meaning is straightforward and intuitive in this context: It simply means that the value of interest (in our case, temperature) at any point is the **average of the neighbor's values**.

The Serial Code

Let's represent the temperate points on the metal plate as a typical **two-dimensional array**.

There are many ways you can process this simulation, so let's pick a straightforward one: **Jacobi iteration**.This method simply says that if you go over your grid and set each element equal to the average of the neighbors, and keep doing this, you will eventually converge on a steady state. Of course, these are floating-point numbers, so you will pick some small error, which defines “nothing happening.” In this case, we will say that when **no element changes by more than one-hundredth of a degree**, we are done. We’ll define this threshold as a parameter that we can tune later.

Thus, the core of your algorithm will be something like this loop:

```
// Run once per timestep
for(i = 1; i <= HEIGHT; i++) {
    for(j = 1; j <= WIDTH; j++) {
        temp[i][j] = 0.25 * ( temp_prev[i+1][j] +
            temp_prev[i-1][j] + temp_prev[i][j+1] +
            temp_prev[i][j-1] );
    }
}
```

Remember, you are iterating over your metal plate in both dimensions and setting every interior point equal to the average of the neighbors (i.e., adding together and dividing by 4). **You don’t change the very outside elements;** those are the "heating elements" (or boundary conditions). There are a few other items to go through in the outer iteration loop:

```
while ( worst_dt > TEMP_TOLERANCE ) {
    for(i = 1; i <= HEIGHT; i++) {
        for(j = 1; j <= WIDTH; j++) {
            temp[i][j] = 0.25 * ( temp_prev[i+1][j] +
                temp_prev[i-1][j] + temp_prev[i][j+1] +
                temp_prev[i][j-1] );
        }
    }
    worst_dt = 0.0;
    for(i = 1; i <= HEIGHT; i++){
        for(j = 1; j <= WIDTH; j++){
            worst_dt = fmax(
                fabs(temp[i][j] - temp_prev[i][j]),
                worst_dt);
            temp_prev[i][j] = temp[i][j];
        }
    }
    if((iteration % 100) == 0)
        track_progress(iteration);
    iteration++;
}
```

The important addition is that you have a **second array that keeps the temperature data from the last iteration**: using a single array will cause **race conditions** as you update the temperature from one thread while another is reading it to calculate the average. We’ll copy over from one array to another in a separate loop, which in the meantime **calculates the error**.

It might also be nice to track your progress as you go; every 100 iterations, let's call a modest output routine which prints some data points from the last row.

The complete serial implementation is given in `laplace-seq.cpp`.

- Compile the code in a terminal:

```
$ g++ -Ofast -march=cascadelake -o laplace-seq laplace-seq.cpp
```

- Let's run our program on a 3.90GHz Intel Xeon Gold CPU node:

```
$ srun -C xcne -- ./laplace-seq
```

Exercise 9

Study the code in `laplace-seq.cpp` carefully. **Come up with some optimizations using all the OpenACC constructs** you have learned so far: focus on optimizing the main loop nest. Make a copy of `laplace-seq.cpp`, implement your ideas and compile it for the **H-100 GPU**.

You can compare your program with the provided `laplace-acc` binary; **try to beat it** in terms of runtime performance, without referring to the `laplace-acc.cpp` source file.

Once you've done all you can, turn over the page to see the optimizations used in `laplace-acc`.

Hint 1: It may not be wise to apply a loop construct to the outer iteration loop.

Hint 2: You will probably need a reduction somewhere.

Hint 3: Use profilers like `nsys nvprof` to get an idea of the hotpaths in your code, such as the most common and expensive API calls.

Hint 4: It would help to step through the code and think about the data movements between host and device. Some of them may be redundant.

[Blank Page]

The Parallel Code: Naïve Version

The first thing you may be tempted to do is to put `parallel loop` directives in front of the two big loops:

```
while ( worst_dt > TEMP_TOLERANCE ) {
    #pragma acc parallel loop collapse(2)
    for(i = 1; i <= HEIGHT; i++) {
        for(j = 1; j <= WIDTH; j++) {
            temp[i][j] = 0.25 * ( temp_prev[i+1][j] +
                temp_prev[i-1][j] + temp_prev[i][j+1] +
                temp_prev[i][j-1] );
        }
    }
    worst_dt = 0.0;
    #pragma acc parallel loop collapse(2) reduction(max:worst_dt)
    for(i = 1; i <= HEIGHT; i++){
        for(j = 1; j <= WIDTH; j++){
            worst_dt = fmax(
                fabs(temp[i][j] - temp_prev[i][j]),
                worst_dt);
            temp_prev[i][j] = temp[i][j];
        }
    }
    if((iteration % 100) == 0)
        track_progress(iteration);
    iteration++;
}
```

Compiling this, you see "`..., #pragma acc loop gang  blockIdx.x` " messages for both of your loops. You may even notice that a reduction was automatically generated for `worst_dt` if you didn't already write it yourself. So far so good.

If you run the executable, you will get something like this:

```
. . .
---------- Iteration number: 3200 ------------
. . .[640,1024]: 61.25  [768,1024]: 73.50  [896,1024]: 85.76  [1024,1024]: 99.87
---------- Iteration number: 3300 ------------
. . .[640,1024]: 61.27  [768,1024]: 73.53  [896,1024]: 85.78  [1024,1024]: 99.87

Max error at iteration 3375 was 0.009999
Total time was 6897.079590 ms.
```

As a reference, the sequential variant typically takes **$\tilde{4}$.7 seconds** to complete. We've moved from running on a single thread to a Nvidia H100 with something on the order of 270,000 threads. For all our efforts, we have managed to slow down the code by about 70 percent, which is less than impressive.

Why did our code slow down? The first suspect that comes to mind for any experienced GPU programmer is **data movement**. The device-to-host memory bottleneck is usually the culprit for such a disastrous performance result as this.

OpenACC: Basic Profiling

To cut a long story short, profiling the application by running `srun -G h100-96 nsys nvprof ./laplace` gives you additional output:

```
   . . .
 Time (%)  Total Time (ns)  Num Calls  Avg (ns)   ...          Name
 --------  ---------------  ---------  ---------  ...  --------------------
     80.8    5,441,224,669     13,504  402,934.3  ...  cuMemcpy2DAsync_v2
     14.4      972,068,055      3,376  287,934.9  ...  cuMemcpyHtoDAsync_v2
      3.6      239,441,372     20,256   11,820.8  ...  cuStreamSynchronize
      0.5       36,491,624     10,128    3,603.0  ...  cuLaunchKernel
   . . .
```

The problem is not subtle. From the compiler output, each of the `parallel loop` constructs copies any data used within them to the device at the beginning of the data region, and copys them back to the host at the end. Because each of these constructs are themselves in a non-parallelized loop, this results in **two large arrays getting copied back and forth twice for each iteration of the main loop**:

```
while ( worst_dt > TEMP_TOLERANCE ) {
                                     // In each of 3000 iterations...
    #pragma acc parallel loop        // copyin(temp,temp_prev)
    for(i = 1; i <= HEIGHT; i++) {
        ...
    }                                // copyout(temp,temp_prev)
    worst_dt = 0.0;
    #pragma acc parallel             // copyin(temp,temp_prev)
    for(i = 1; i <= HEIGHT; i++){
        ...
    }                                // copyout(temp,temp_prev)
    ...
}
```

These are two 1,026 × 1,026 double-precision arrays, so this amounts to (2 arrays) × (1,026 × 1,026 points/array) × (8 bytes/point) = **16MB of memory copies per iteration**.

The Parallel Code: Optimized Version

So far we have marked the parallel regions for acceleration. Now it is time to introduce data regions to **optimize data transfers**.

Pause here and see whether you can come up with a strategy to minimize data movement. What directives does that strategy translate to?

If you load your data onto the device at the beginning of the main loop, when do you next need it on the host? Think the first iteration through as a start: there is **no reason for the two big arrays to return to the host in between the two loops**. They can stay on the device. At the start of the second iteration, you would like both big arrays to be on the device, and that is just where you left them! **So it looks as if you can just keep the data on the device between iterations of the outer while loop**. The obvious way to implement this would be a `data copy clause` bracketing the while loop, decoupling the data and execution regions.

```
#pragma acc data copy(temp_prev, temp)
{
    while ( worst_dt > TEMP_TOLERANCE ) {
. . .
```

We can't call it a day just yet: there is now a bug with the `track_progess()` output routine that gets invoked every 100 iterations printing all zeros. You still need for the temperature data to be copied back on the host at that point: otherwise, the host copy of `temp` will remain at the initial condition until the data copy happens at the termination of the while loop, at the very end of the data region.

To fix this, we just apply an `update`:

```
. . .
if((iteration % 100) == 0) {
   #pragma acc update host(temp)
   track_progress(iteration);
}
. . .
```

Extra Tweaks

There's one more tweak you can apply to the code, once you understand that **you don't actually need to copy both big arrays into the `while` loop**. It happens that `temp_prev` is the array that is initialized in the initialization routine, and `temp` uses these values to initialize itself in the first iteration. By the same reasoning, since you no longer access either of the arrays once you exit the while loop, no copy-out is needed.

This means that `temp` can aptly just be a temporary array used on the device. **We'll `copyin` the `temp_prev` array and just allocate memory for `temp` with a `create` clause.**

Final result

```c
#pragma acc data copyin(temp_prev), create(temp)
{
    while ( worst_dt > TEMP_TOLERANCE ) {
        #pragma acc parallel loop collapse(2)
        for(i = 1; i <= HEIGHT; i++)
            for(j = 1; j <= WIDTH; j++)
                temp[i][j] = 0.25 * ( temp_prev[i+1][j] +
                    temp_prev[i-1][j] + temp_prev[i][j+1] +
                    temp_prev[i][j-1] );

        worst_dt = 0.0;
        #pragma acc parallel loop collapse(2) \
            reduction(max:worst_dt)
        for(i = 1; i <= HEIGHT; i++){
            for(j = 1; j <= WIDTH; j++){
                worst_dt = fmax(
                    fabs(temp[i][j] - temp_prev[i][j]),
                    worst_dt);
                temp_prev[i][j] = temp[i][j];
            }
        }
        if((iteration % 100) == 0)
            track_progress(iteration);
        iteration++;
    }
}
```

```
. . .
---------- Iteration number: 3200 ------------
. . .[640,1024]: 61.25  [768,1024]: 73.50  [896,1024]: 85.76  [1024,1024]: 99.87
---------- Iteration number: 3300 ------------
. . .[640,1024]: 61.27  [768,1024]: 73.53  [896,1024]: 85.78  [1024,1024]: 99.87

Max error at iteration 3375 was 0.009999
Total time was 505.544006 ms.
```

This is much better! Roughly a 9× improvement over the serial code.

Exercise 10
There are, of course, still many more ways we can optimize this program; try to explore further! Given what you know so far, **see if you can beat the preformance of** `laplace-acc-opt`.

Summary

With what you've learned in this tutorial, you will hopefully be able to write parallel programs in an abstract machine model and simultaneously target various types of processors. You should also have available a range of fundamental directives for developing and optimizing functioning parallel programs. You can run code on an accelerator, and you know how to use various mechanisms to reduce the data traffic required when running on distributed memory machines.
However, the tools presented in this tutorial barely scratches the surface of OpenACC's features. For more information on

- compiler-driven optimization using `kernels`,
- `async` constructs and work-queue synchronization,
- heterogeneous multi-device programming, and
- advanced performance profiling,

... please refer to the resources in the appendix.

Appendix: OpenACC / Performance Resources

Resources

- For more details on OpenACC constructs, please refer to the OpenACC documentation at

  https://www.openacc.org/sites/default/files/inline-files/APIGuide2.7.pdf

- For more in-depth programming guidance, refer to

  https://openacc-best-practices-guide.readthedocs.io/en/latest/

- The GCC compiler supports OpenACC as well, with a few caveats. You can try installing and using an OpenACC-enabled offloading GCC runtime through Spack:

  https://www.openacc.org/tools/gcc-for-openacc

  For more information:

  https://gcc.gnu.org/wiki/OpenACC/Quick%20Reference%20Guide

- The main reference for this tutorial, including the source of some of the examples, is the official textbook. Please approach your tutor if you're interested in obtaining a copy:

  https://www.informit.com/store/openacc-for-programmers-concepts-and-strategies-9780134694283