# Toward Automatic Translation: From OpenACC to OpenMP 4

by

Nawrin Sultana

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 10, 2016

Keywords: GPUs, OpenACC, OpenMP, Translation

## Abstract

For the past few years, OpenACC has been the primary directive-based API for programming accelerator devices like GPUs. OpenMP 4.0 is now a competitor in this space, with support from different vendors. In our work, we analyse the feasibility for automatic conversion from OpenACC to OpenMP 4. We describe an algorithm to convert OpenACC device directives to OpenMP 4; we implemented this algorithm in a prototype tool and evaluated it by translating the EPCC Level 1 OpenACC benchmarks. We discuss some of the challenges in the conversion process and propose what parts of the process should be automated, what should be done manually by the programmer, and what future research and development is necessary in this area.

## Acknowledgments

I would like to take this opportunity to acknowledge all who helped me in completing my research.

Firstly, I would like to thank my supervisor Dr. Jeffrey Overbey for his excellent insights and constant supervision on my research. His ambition and passion for the subject inspired my motivation. I highly appreciate his extreme attention to detail and continuous feedback. This work would not have been possible without his constant support and dedication.

I would also like to thank my other committee members, Dr. Anthony Skjellum and Dr. James Cross, for their support and time to evaluate my research and helping me to graduate from the University.

I would like to express my deepest gratitude to Galen Arnold from National Center for Supercomputing Applications for his significant contribution into this work by providing useful suggestions and resources.

Finally, I thank my fellow student, Alexander Calvert for working with and beside me during my research.

Table of Contents

List of Figures

# Chapter 1
# Introduction

Directive-based programming of graphics processing units (GPUs) is an emergent technique as an alternative to using the canonical APIs such as CUDA and OpenCL for programming accelerator devices. Scientific programmers are increasingly turning to directive-based APIs as they are more concise and declarative. OpenACC has been the standard directive-based API for scientific programmers targeting NVIDIA GPUs, since it first appeared in 2011. In 2013, OpenMP 4.0 added directives for accelerators and with much similarity to OpenACC. Both OpenMP 4 and OpenACC have data directives and clauses, and parallelism directives.

## 1.1 Motivation

Support of OpenACC and OpenMP is available in commercial compilers from PGI, Cray, and GCC. Systems like—x86 plus Intel Xeon Phi (Knights) have OpenMP 4 compiler support while x86 plus NVIDIA GPU or AMD GPU have OpenACC compiler support. Programmer want to run a code across both systems need two slightly different programs.

The goal is to be able to make a single source base possible for different accelerators. And the availability of coherent OpenMP and OpenACC implementations is a problem.

Since nearly every compiler supports OpenMP for multicore parallelism, it is reasonable to expect that the device directives added in OpenMP 4 will eventually supplant OpenACC. However, that certainly has not happened yet, and given the strong backing for OpenACC from PGI/NVIDIA, there may continue to be some rivalry between the two specifications. But Cray recently announced that their support for OpenACC will be frozen at version 2.0, and future development will focus on OpenMP 4 device directives.

In our work, we develop a source-to-source program conversion tool, from OpenACC to OpenMP 4. For systems that have OpenMP but do not have OpenACC support yet, this conversion tool could be very useful.

## 1.2 Thesis Statement and Contributions

Since OpenACC inspired the device directives added in OpenMP 4.0, the two APIs are quite similar. In many cases, there is a 1–1 mapping between directives. However, there are some important differences as well.

Our work describes an early effort in automatically translating OpenACC directives into OpenMP 4 directives for C codes. We make the following contributions:

- We investigate the feasibility of a tool to semi-automatically convert OpenACC code to use OpenMP 4 device directivess.
- We prototype a tool that automates a large part of the conversion process.
- We evaluate our tool using the EPCC Level 1 benchmarks, along with a few miscellaneous examples.
- We identify parts of the conversion process where manual intervention is essential.

### 1.2.1 Target Hardware

We focused on converting OpenACC to OpenMP 4 *that targets the same accelerator device*—a Tesla k40 NVIDIA GPU. We used PGI C/C++ 16.1 as our OpenACC compiler, and we used Clang 3.8 as our OpenMP compiler.

We chose this setup because it allowed us to investigate changes in semantics and performance that result from *changing the compiler and API*, rather than those that result from a change in target hardware. However, changing the target hardware in addition to the compiler and API introduces additional complexity. Keeping the target hardware fixed allowed us to focus on the differences between OpenACC and OpenMP, rather than differences in the accelerator devices themselves.

## 1.3 Thesis Outline

The thesis is outlined as follows:

- Chapter 2 gives a brief history and overview of OpenACC and OpenMP.
- Chapter 3 discusses translation from OpenACC to OpenMP, focusing on complications that prohibit completely automatic translation.
- Chapter 4 describes our tool and the algorithm it uses to convert OpenACC to OpenMP device directives.
- Chapter 5 presents empirical data describing the conversion.
- Chapter 6 discusses related work.
- Chapter 7 describes limitations of the current tool and directions for future work.

# Chapter 2

# Background

## 2.1 History of OpenACC and OpenMP

OpenACC has a short history. It was originally developed by PGI, Cray, and NVIDIA, with version 1.0 being released in 2011. Version 2.0 was released in 2013, adding support for procedure calls and nested parallelization (enabled by new CUDA capabilities), among other features. Version 2.5 was a more minor revision, made in 2015 [1].

OpenMP dates back to the late 1990s, when the OpenMP Architecture Review Board was formed to develop a specification for directive-based parallelism that could be used commonly across multiple vendors' multiprocessors. The first OpenMP specification was released for Fortran in 1997 and for C/C++ in 1998. The OpenMP specification has undergone several revisions since that time. Notably, version 4.0 added support for accelerators in 2013; an update to version 4.5 followd in 2015 [2]. The so-called *device directives* or *target directives* added to support this were based on OpenACC. However, as we will see, the two APIs are not identical.

## 2.2 CUDA Execution Model

OpenACC and OpenMP device directives are both designed to offload computations to accelerator devices—massively parallel devices like GPUs and Xeon Phis, designed for data parallel computations. Each API has its own execution model, which is intentionally abstract to avoid coupling it to the specifics of any one device.

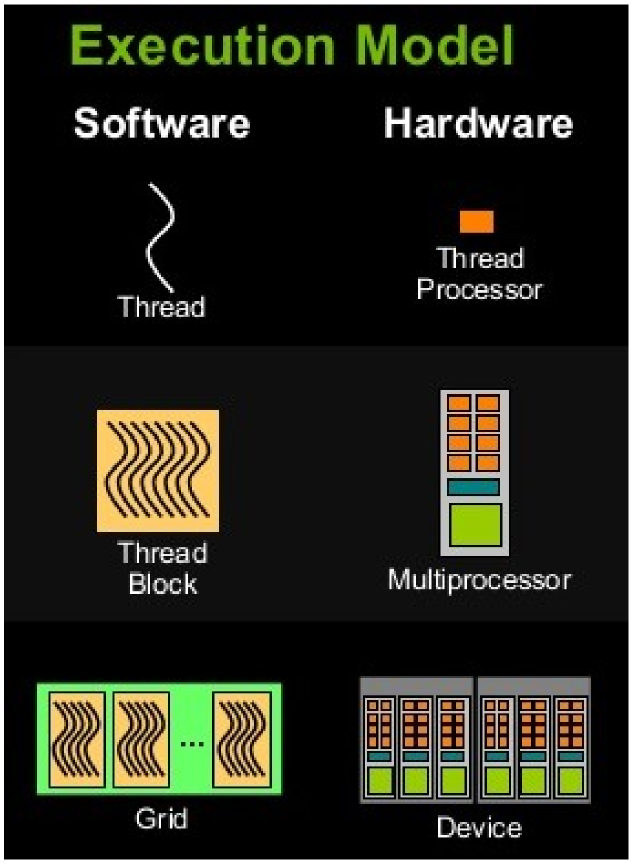


Figure 2.1: CUDA Execution Model [3]

For the purposes of this work, we will only focus on one type of accelerator device: NVIDIA GPUs. To understand the behavior of OpenACC and OpenMP constructs on these devices, it is helpful to understand their execution model—the CUDA execution model.

CUDA-capable devices (i.e., NVIDIA GPUs) execute *kernels*, functions that execute on the device by running many GPU threads concurrently. Only one kernel executes at a time. All of the threads in a kernel are collectively called a *grid*. A grid is comprised of one or more *thread blocks*, and each thread block is comprised of one or more *threads*.

In hardware, each thread block is assigned to one Streaming Multiprocessor (SM). Each SM consists of several Streaming Processors (SPs); each thread of a thread block is assigned to one of the SPs within the SM. Thread blocks are divided into 32-thread groups called *warps*. Instructions are fetched and executed on a per-warp basis, not on a per-thread basis, so the 32 threads within each warp execute instructions in single instruction, multiple data

(SIMD) fashion. The SM context switches from one warp to another at every instruction issue ("zero-overhead thread scheduling"), so while threads within a warp execute in lockstep, threads in *different* warps may not. Threads within a block can coordinate using barrier synchronization, whereas threads in different blocks cannot [9].

## 2.3 OpenMP 4 Directives

OpenMP directives for C/C++ are specified with the *#pragma* preprocessing directive. The syntax of an OpenMP directive is informally specified as follows:

`#pragma omp` *directive-name [clause[[,] clause]. . . ] new-line*

Each directive starts with `#pragma omp`. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. Some OpenMP directives may be composed of consecutive `#pragma` preprocessing directives if specified in their syntax. Directives are case-sensitive. An OpenMP executable directive applies to at most one succeeding statement, which must be a standard block.

The following are the most commonly used OpenMP directives:

***Parallel.*** OpenMP's fundamental construct is the parallel construct. When a parallel construct is encountered by a thread, that thread creates a *team* of new threads, becoming that team's master. The threads are assigned numbers, with the master given number 0. The threads each execute a copy of the parallel region's code. Once the *team* is created, the number of threads in the team remains constant for the duration of that *parallel* region. The syntax of **parallel** construct is as follows:

`#pragma omp parallel` *[clause[[,] clause]. . . ] new-line*

where *clause* is one of the following:

**if**(scalar-expression)

**num_threads**(integer-expression)

**default**(**shared** | **none**)

**private**(list)

**firstprivate**(list)

**shared**(list)

**copyin**(list)

**reduction**(reduction-identifier : list)

**proc_bind**(**master** | **close** | **spread**)

***Loop.*** Inside a parallel region, a *for* construct may be specified. A for region is associated with a for loop, and upon encountering the region, a different thread from the current team is assigned to each iteration of the for loop. A parallel for region is also defined as a shorthand for a parallel region containing nothing but a single for region. OpenMP 4.0 adds a similar construct, simd, which divides work of the associated loop up among SIMD lanes of the current thread.

The syntax of **loop** construct is as follows:

```
#pragma omp for [clause[[,] clause]...] new-line
```

where *clause* is one of the following:

**private**(list)

**firstprivate**(list)

**lastprivate**(list)

**reduction**(reduction-identifier : list)

**schedule**(kind[, chunk_size)

**collapse**(n)

**ordered**

**nowait**

All associated *for-loops* must have *canonical loop* form.

***Target Data.*** A target data region may be specified as well. A target data region creates a data environment for execution. It may specify a device clause, which specifies a device to execute the environment on, and a map clause, which specifies a mapping of data from the host to the device. The syntax of **target data** construct is as follows:

```
#pragma omp target data [clause[[,] clause]...] new-line
```

where *clause* is one of the following:

**device**(integer-expression)

**map**([map-type :] list)

**if**(scalar-expression)

***Target Teams.*** A target teams construct is equivalent to a target construct containing a teams construct. The target teams construct creates a data environment and a *league* of thread teams to execute it. When a thread encounters a *teams* construct, a league of thread teams is created and the master thread of each thread team executes the *teams* region.

The syntax of **teams** construct is as follows:

```
#pragma omp teams [clause[[,] clause]...] new-line
```

where *clause* is one of the following:

**num_teams**(integer-expression)

**thread_limit**(integer-expression)

**default**(**shared** | **none**)

**private**(list)

**firstprivate**(list)

**shared**(list)

**reduction**(reduction-identifier : list)

***Distribute.*** A distribute construct creates a league of teams and specifies that contained loops will be executed by those teams. It is associated with a loop nest consisting of one or more loops that follow the directive.

The syntax of **distribute** construct is as follows:

```
#pragma omp distribute [clause[[,] clause]...] new-line
```

where *clause* is one of the following:

**private**(list)

**firstprivate**(list)

**clooapse**(n)

**dist_schedule**(kind[,chunk_size])

***Distribute Parallel Loop.*** A distribute parallel for construct combines distribute, parallel, and for construct to specify that a loop can be executed in parallel by multiple threads in multiple teams. The effect of any clause that applies to both the distribute and parallel loop constructs is as if it were applied to both constructs separately. The syntax of **distribute parallel loop** construct is as follows:

```
#pragma omp distribute parallel for [clause[[,] clause]...] new-line
```

where *clause* can be any of the clauses accepted by *distribute* or *parallel loop* directives.

## 2.4 OpenACC Directives

OpenACC has fewer constructs. This is in part because OpenACC is specifically for use with massively parallel accelerator hardware, where OpenMP supports accelerators as well as multicore CPUs. However, many of OpenACC's construct are analogous to OpenMP ones.

In C/C++, OpenACC directives are specified with the `#pragma` mechanism. The syntax of an OpenACC directive is:

```
#pragma acc directive-name [clause[[,] clause]...] new-line
```

Each directive starts with `#pragma acc`. The remainder of the directive follows the C and C++ conventions for pragmas. Directives are case-sensitive. An OpenACC directive applies to the immediately following statement, structured block or loop.

Following are some of the commonly used OpenACC constructs:

***Parallel.*** OpenACC also has a parallel construct. A parallel construct, when encountered, creates *gangs of workers* to execute the region on the accelerator.One worker in each gang begins executing the code in the structured block of the construct. The syntax of OpenACC **parallel** directive is:

```
#pragma acc parallel [clause[[,] clause]...] new-line
        structured block
```

where *clause* is one of the following:

**if**(condition)

**async**[(scalar-integer-expression)]

**num_gangs**(scalar-integer-expression)

**num_workers**(scalar-integer-expression)

**vector_length**(scalar-integer-expression)

**reduction**(operator : list)

**copy**(list) | **copyin**(list) | **copyout**(list) | **create**(list)

**present**(list)

**present_or_copy**(list)

**present_or_copyin**(list)

**present_or_copyout**(list)

**present_or_create**(list)

**deviceptr**(list)

**private**(list)

**firstprivate**(list)

***Kernels.*** The kernels construct specifies a region that is to be divided into a sequence of kernels for accelerator execution. The loop and shorthand kernels loop are available for kernels directives as they are for parallel directives.

The syntax of OpenACC **kernels** directive is:

```
#pragma acc kernels [clause[[,] clause]...] new-line
            structured block
```

where *clause* can be any of the clauses accepted by *parallel* construct except *num_gangs*, *num_workers*, *vector_length*, *reduction*, *private*, and *firstprivate*.

***Loop.*** A loop construct is also available, specifying the type of parallelism for the associated loop. There is also the parallel loop abbreviation, a shorthand for a loop construct immediately inside a parallel construct.

The syntax of OpenACC **loop** directive is:

```
#pragma acc loop [clause[[,] clause]...] new-line           for loop
```

where *clause* is one of the following:

**collapse**(n)

**gang**[(scalar-integer-expression)]

**worker**[(scalar-integer-expression)]

**vector**[(scalar-integer-expression)]

**seq**

**reduction**(operator : list)

**independent**

**private**(list)

In a parallel region, a loop directive with no *gang*, *worker* or *vector* clause allows compiler to automatically select whether to execute the loop across gangs, workers within a gang, or as vector operations.

***Data.*** A data directive is also defined. It specifies for a region what data should be copied into the accelerator upon entry to the region and copied out on exit and defines what data is to be allocated on the device for the duration of the region.

The syntax of OpenACC **data** directive is:

```
#pragma acc data [clause[[,] clause]...] new-line           structured block
```

where *clause* is one of the following:

**if**(condition)

**copy**(list) | **copyin**(list) | **copyout**(list) | **create**(list)

**present**(list)

**present_or_copy**(list)

**present_or_copyin**(list)

**present_or_copyout**(list)

**present_or_create**(list)

**deviceptr**(list)

A more complete description of these directives is available in the OpenACC specification.

# Chapter 3
# Translation Considerations

Our goal was to create a tool to convert OpenACC code to equivalent OpenMP code, automating as much of the process as possible. For the purposes of prototyping, we focused on translating C code (as opposed to C++ or Fortran).

## 3.1 A Subset of OpenACC

For the purposes of prototyping, we designed our tool to accept the subset of OpenACC 1.0 directives described by the grammar in Figure 3.1. This includes all of the directives and clauses used by the example programs in our test suite (notably, the EPCC Level 1 benchmarks). While it omits some important features—asynchronous execution, unstructured data lifetimes, etc.—it does correspond to a useful, commonly-used subset of OpenACC.

## 3.2 Considerations in Translation

In theory, the goals of our work are modest: We are interested in translating directives in OpenACC 1.0 specification [11] (Figure 3.1) to OpenMP 4, targeting the same accelerator devices (NVIDIA GPUs). In theory, the two APIs are very closely aligned, and the translation should be straightforward.

In practice, the situation is not so simple. The OpenACC and OpenMP specifications define an *abstract* execution model. They do not dictate how that model is mapped to specific hardware devices. This is left to the compiler. Our OpenACC programs were compiled using PGCC 16.1, the standard commercial compiler for OpenACC. We compiled OpenMP 4 using a custom build of Clang 3.8 [12] (since PGCC does not yet support OpenMP 4). Again

```
acc-directive  →  #pragma acc data data-clauses
               |  #pragma acc parallel par-clauses
               |  #pragma acc loop type loop-clauses
               |  #pragma acc update update-clauses
         type  →  gang vector
               |  gang
               |  vector
               |  seq
  data-clause  →  copyin(vars)
               |  copyout(vars)
               |  copy(vars)
               |  create(vars)
               |  if(condition)
               |  present(vars)
               |  present_or_copy(vars)
               |  present_or_copyin(vars)
               |  present_or_copyout(vars)
               |  present_or_create(vars)
   par-clause  →  num_gangs(n)
               |  vector_length(n)
               |  private(vars)
               |  firstprivate(vars)
               |  reduction(reductions)
               |  data-clause
  loop-clause  →  independent
               |  private(vars)
               |  reduction(reductions)
               |  collapse(n)
update-clause  →  host(vars)
               |  device(vars)
               |  if(condition)
```

Figure 3.1: OpenACC subset grammar accepted by the translation tool.

```
#pragma acc data copyin(A[0:N], B[0:N]), copyout(C[0:N])
#pragma omp target data map(to:A[0:N], B[0:N]), map(from:C[0:N])
{
     #pragma acc parallel loop
     #pragma omp target teams distribute parallel for
     for (int i = 0; i < N; ++i) {
         C[i] = A[i] + B[i];
    }
}
```

Figure 3.2: Vector addition kernel.

Cray's compilers support both OpenACC and OpenMP. We compiled our test programs against Cray's compilers. As we will show, directives that appear similar may be translated quite differently by the two compilers.

### 3.2.1 Work Distribution for Single Loop

Perhaps the simplest example of directive-based accelerator code is the vector addition kernel shown in Figure 3.2. It adds two vectors $A$ and $B$ on an accelerator device, storing the sum in $C$. While the directives direct the compiler to parallelize the loop on the accelerator, the compiler can decide (in this case) how to divide the loop iterations among threads and thread blocks on the CUDA device. When $N = 10000$:

- PGCC generates a grid of 79 thread blocks with 128 threads per block; kernel execution time is about 3 $\mu s$ on a Tesla K40.

- Clang generates a grid of 15 thread blocks with 1024 threads per block; kernel execution time is about 30 $\mu s$ on a Tesla K40. Adding `num_teams(79) thread_limit(128) to the OpenMP code (to match the grid and block dimensions of the OpenACC code) actually increases the kernel execution time to about` 80 $\mu s$.

It is also worth noting that PGCC generates code where all threads are active in all thread blocks except the last (presumably to minimize control divergence). Clang generates code that assigns an approximately equal number of threads to each thread block, so inactive threads may be present in several blocks.

### 3.2.2 Work Distribution for Loop Nests

More important is the handling of loop nests, such as the simple doubly nested loops in Figure 3.3(a).

In OpenACC, the compiler has some freedom to decide how to parallelize the loop nest. PGCC will typically run the outermost loop as a gang loop and the innermost loop as a vector loop. Given the OpenACC code in Figure 3.3(a), the compiler does exactly this: the iterations of the $i$-loop are divided among the thread blocks, and the iterations of the $j$-loop are divided among the threads within each block. The OpenACC code in Figure 3.3(b) is functionally equivalent but makes this distribution explicit.

In OpenMP, work distribution is not so automatic. Consider these "obvious" but incorrect translations: The `#pragma acc parallel loop` directive could be replaced by:

- `#pragma omp target teams`. In this case, the entire loop nest would be run redundantly by all thread blocks.
- `#pragma omp target teams distribute`. In this case, the iterations of the $i$-loop would be divided among thread blocks, but only one thread would be active within each block.
- `#pragma omp target teams distribute parallel for`. Here, the iterations of the $i$-loop are distributed among threads and thread blocks.

In all three cases, *the inner $j$-loop runs sequentially*, because the parallelization directive is applied to only one loop. OpenMP requires the programmer to explicitly specify how to run each loop in the nest.

The translation most similar to the OpenACC code is shown in Figure 3.3(c). The iterations of the $i$-loop are divided among thread blocks using the `distribute` directive, and the iterations of the $j$-loop are divided among threads in the block using the `parallel` `for` directive. OpenACC's `gang` clause is roughly equivalent to OpenMP's `distribute`: effectively, it distributes work among thread blocks. Likewise, OpenACC's `vector` clause is roughly equivalent to OpenMP's `parallel for` inside a `teams` region: it distributes work among threads within a block.

### 3.2.3 Reductions Differ Across Compilers

In a nested loop, reduction may behave differently in different compiler. Consider the triply nested loop in Figure 3.4. When input array "data" is initialized to {1, 10, 100, 1000, 10000}:

- PGCC and Cray compilers produce {1, 11, 111, 1111, 11111} which is similar to the serial output.
- Clang produces {1, 12, 123, 1234, 12345} as output.

We see that in Clang compiler, because of the *reduction* on *tmp* in the inner-most loop the variable `tmp` never gets initialized to 0. But according to OpenMP specification, a private copy should be created and initialized with the initializer value of the *reduction-identifier* in each implicit task or SIMD lane.

## 3.3 Data Transfer

In both OpenACC and OpenMP, if `data/target data` directives are omitted, the compiler will automatically determine what data needs to be transferred between the host and the accelerator. This is another case where compilers may differ. In the vector addition example (Figure 3.2), consider what happens when $A$, $B$, and $C$ are *pointers* allocated just before the kernel (in the same function) using *malloc*. With PGCC, the `#pragma acc data` directive can be omitted; it can determine statically that 10,000 elements of $A$, $B$, and $C$ are accessed and thus need to be copied. With Clang, if the `#pragma omp target data` directive is omitted, the kernel crashes at runtime (*device illegal address*).

*OpenACC (Original):*

```
#pragma acc parallel loop
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        ...
    }
}
```

(a)

*OpenACC (Equivalent):*

```
#pragma acc parallel loop gang
for (int i = 0; i < 10; i++) {
    #pragma acc parallel loop vector
    for (int j = 0; j < 10; j++) {
        ...
    }
}
```

(b)

*OpenMP 4:*

```
#pragma omp target teams distribute
for (int i = 0; i < 10; i++) {
    #pragma omp parallel for
    for (int j = 0; j < 10; j++) {
        ...
    }
}
```

(c)

Figure 3.3: Doubly nested loops.

```
#pragma acc parallel loop gang copyout(result) copyin(data)
#pragma omp target teams distribute map(from:result) map(to:data)
for (int i = 0; i < 1; i++) {
    for (int j = 0; i < 5; j++) {
         int tmp = 0;
        #pragma acc loop vector reduction(+:tmp)
        #pragma omp parallel for reduction(+:tmp)
        for (int k = 0; k <= j; k++) {
            tmp += data[k];
        }
        result[j] = tmp;
    }
}
```

Figure 3.4: Reduction in nested loop.

# Chapter 4
# Translation Algorithm and Tool Implementation

In the previous chapter, we showed that OpenACC and OpenMP 4 device directives handle work distribution differently, and that differences in compilers can have significant effects on the performance (and correctness) of code. This means that porting from OpenACC to OpenMP 4 will almost certainly involve some manual effort. Nevertheless, much of the translation is straightforward and tedious, which makes it worth automating.

Therefore, we designed our translation tool—and its underlying algorithm—with the following objectives:

- *The translation should be straightfoward and consistent.* Each OpenACC directive should translate to the same OpenMP directive(s) every time, and the translation should mirror the structure of the OpenACC code, to the extent possible.

- *Automated translation and manual editing should be complementary.* The tool should not try to automate too much. It should not perform loop nest optimizations; it should not infer implicit data transfers; it should not make any invasive code changes. The tool should automate the tedious, most obvious parts of the translation, in a completely predictable way. More sophisticated changes—like performance tuning—should be under the control of the programmer.

The translation algorithm is as follows. The input is a C source file containing OpenACC directives. The output is the same file with OpenMP directives substituted.

1. If any of the following occur, warn the user; these structures will not be automatically converted:

   - The header file `openacc.h` is included.
   - The macro `_OPENACC` is used.
   - Functions in the OpenACC runtime library are called or referenced.
   - The `kernels` directive is used.

2. Prepare parallel loop nests using the algorithm described in §4.1 below.

3. Apply the translation rules described below in §4.3 to convert OpenACC directives to OpenMP directives.

The `#pragma acc kernels` directive is effectively a request for automatic parallelization; the compiler must decide what loops to parallelize and how. There is no direct equivalent in OpenMP. Thus, our algorithm does not support the kernels directive. Instead, the programmer should modify the code to explicitly parallelize loops using the `parallel` directive.

## 4.1 Loop Nest Preparation

As noted in §3.2.2 above, OpenACC-to-OpenMP translation is complicated by the fact that work distribution is explicit in OpenMP, while it may be left to the compiler in OpenACC. We noted that the code in Figure 3.3(a) is equivalent to that in Figure 3.3(b), expect the latter makes the work distribution explicit.

Before our translation tool attempts to translate any OpenACC directives, it "prepares" each parallel loop nest as follows.

- If the loop nest contains explicit gang and vector clauses, there must be exactly one gang loop and exactly one vector loop in the nest. The same loop may be both the gang and vector loop.

- If the loop nest does not contain an explicit gang loop, the outermost non-sequential loop is chosen as the gang loop.

- If the loop nest does not contain an explicit vector loop, the innermost non-sequential loop is chosen as the vector loop.
- All loops in the loop nest other than the gang and vector loop are run sequentially.
- If a loop will be run sequentially, and it contains a `private` clause, all `private` variables must not be live at the top of the loop.
- If the loop nest does not have the form above (e.g., if it contains two gang loops, or if a vector loop appears outside a gang loop), the translation to OpenMP cannot proceed.

The last rule is necessary because OpenACC is surprisingly lax about where the nesting of gang and vector loops. For example, a `gang` loop may be nested inside a `vector` loop; there is no direct equivalent of this in OpenMP. Moreover, PGCC may ignore the `gang` and `vector` directives altogether, e.g., when a gang loop appears inside another gang loop, or a vector loop appears outside another vector loop.

The next-to-last rule ensures that, if a loop is run sequentially, the `private` clause can be removed without changing the behavior of the loop. Although the rule is stated in terms of a live variables analysis [4], the concept is straightforward. If a variable $a$ is declared as private, but each iteration assigns a value to $a$ before reading its value, then the `private` clause can be eliminated if the loop is run sequentially. It cannot be removed if each iteration of the loop depends on the value of $a$ prior to the loop (e.g., if the first instruction in the loop was `a++`).

The second and third rules implement a heuristic typical of parallelizing compilers (one that PGCC appears to use, in fact): the iterations of the outermost parallelizable loop are partitioned among thread blocks, and the iterations of the innermost parallelizable loop are partitioned among the CUDA threads within that block.

To determine if a loop is parallelizable (for the second and third rules), our tool uses a combination of syntactic checks and dependence analysis [15, 4]:

1. If the loop contains a `seq` clause, it is not parallelizable.

2. If the loop contains a `gang` or `vector` clause, it is parallelizable.

3. Otherwise, we perform a dependence analysis on the loop. If the loop does not carry a dependence, it is parallelizable [4]; if it does, we determine it to be non-parallelizable.

## 4.2 Sequential Private/Reduction Removal

After a loop nest has been prepared, there will be a gang loop, a vector loop (possibly the same loop), and zero or more sequential loops. OpenACC has a directive for sequential loops (`acc loop seq`), which permits *private* and *reduction* clauses. However, OpenMP does not have a sequential loop directive. This means that these directives—and more importantly, *private* and *reduction* clauses—must be removed from sequential loops in a way that will preserve the semantics of the *private* and *reduction* clauses. To remove a *private* clause, we can simply re-declare the privatized variable in the loop body, as shown in Figure 4.1(a) (noting that this features requires C99 support, which is available in all modern C compilers). This declaration shadows the declaration in the outer scope, which gives the intended semantics of the *private* clause. To illustrate removing a *reduction* clause, consider as a concrete example `reduction(+:x)`. There are two cases. *Case 1.* This is perhaps the most commonly occurring use of the *reduction* clause: If the only accesses to `x` in the loop body are in statements of the form `x++` or `x +=` *expression* (where `x` does not occur in *expression*), then the reduction clause can be removed with no change in semantics. This is the case in Figure 4.1(a), for example. *Case 2.* When the above condition does not hold, it is still possible to remove a *reduction* clause. However, preserving the semantics of the *reduction* clause requires a more complex source code transformation, illustrated in Figure 4.1(b). (1) A "fresh" variable (`sum`) is declared to accumulate the result. (2) Since `x` is the reduction variable, a local variable `x` is declared in the loop body with the same type as the shadowed variable `x`, and it is initialized to the initializer value for a sum reduction (`0`). (3) At the end of the loop body, the local `x` is added to the result (`sum`). (4) After the loop, the accumulated result is added to the original `x`.

## 4.3 Directive Translation

The loop nest preparation algorithm in the previous section identifies a single gang loop and a single vector loop in each parallel loop nest, and it guarantees that the vector loop is nested under (or is the same as) the gang loop. After gang, vector, and sequential loops have been identified, it is possible to proceed with translating OpenACC directives to OpenMP directives.

Our tool implements the translation rules in Figure 4.2. The majority of the translation is straightforward: Each phrase in OpenACC is mapped to a corresponding phrase in OpenMP.

It is important to note that, according to the translation rules in Figure 4.2, *each data directive is translated as-is.* No attempt is made to infer implicit data transfers or array lengths.

Also, these rules assume that each loop in a parallel loop nest has been *explicitly* labeled with a `gang`, `vector`, or `seq` clause (or both `gang` and `vector`). Of course, this is not necessary. In fact, our tool runs the loop nest preparation algorithm implicitly; it identifies a gang and vector loop in each parallel loop nest, but it does not literally insert OpenACC `gang`, `vector`, and `seq` clauses into the source code (although it retains them if they are already present in the code).

## 4.4 Translation Tool

We prototyped our OpenACC-to-OpenMP translation algorithm by extending the Eclipse C/C++ Development Tools. In addition to our translation algorithm, we added support for various data flow analyses and dependence analysis, along with a parser for OpenACC directives. Our translation can be performed from within the Eclipse user interface, but we also added a command line interface (CLI), so one can run the translation from the command line, without installing Eclipse. Figure 4.3 presents a screenshot from our tool with source file and the corresponding refactored file.

```
int p, x = 100;
#pragma acc loop seq private(p) reduction(+:x)
for (int i = 0; i < 5; i++) {
     p = 100;
     x += p;
}
```

$$\Downarrow$$

```
int p, x = 100;
for (int i = 0; i < 5; i++) {
     int p;
     p = 100;
     x += p;
}
```

(a)

```
int x = 100;
#pragma acc loop seq reduction(+:x)
for (int i = 0; i < 5; i++)
     x = 1;
```

$$\Downarrow$$

```
int x = 100;
int sum = 0;                 // (1)
for (int i = 0; i < 5; i++) {
     int x = 0;              // (2)
     x = 1;
     sum += x;               // (3)
}
x += sum;                    // (4)
```

(b)

Figure 4.1: Elimination of *private* and *reduction* clauses on a sequential loop.

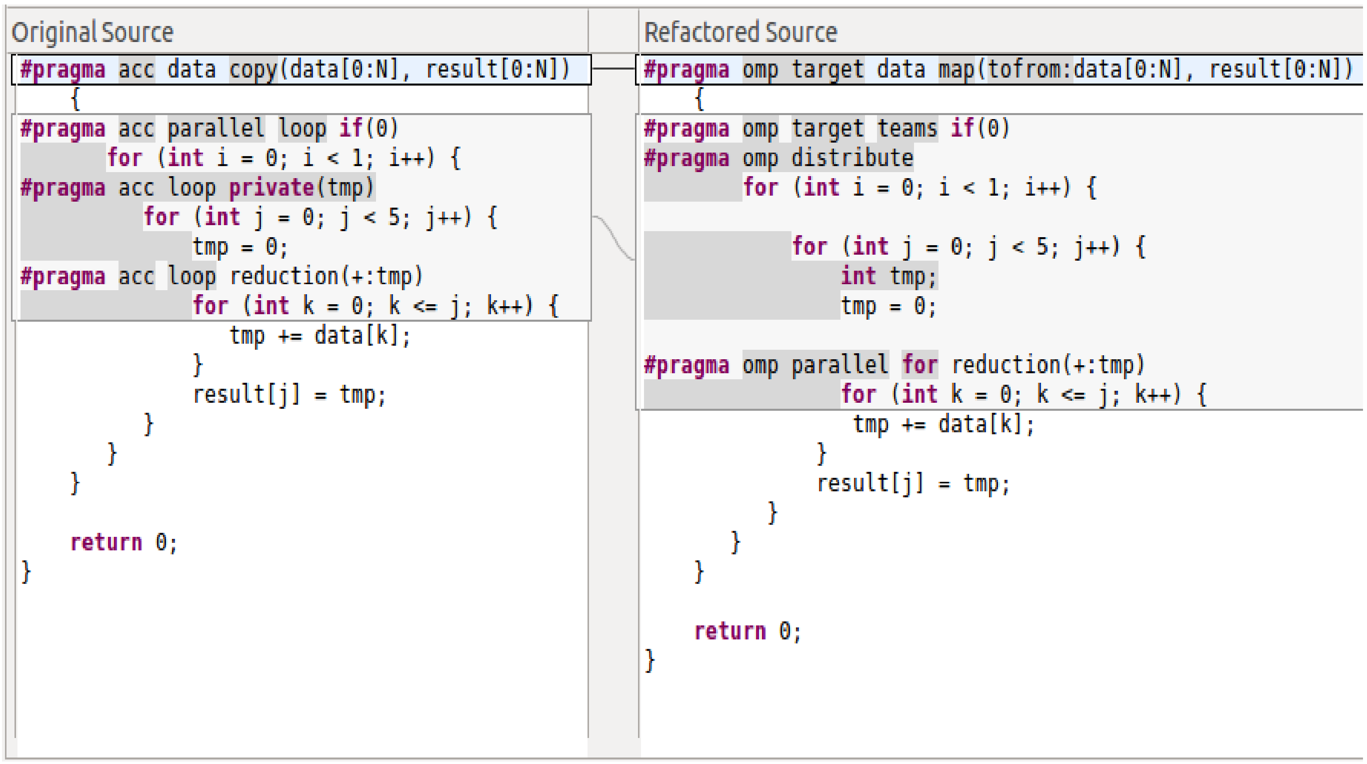
$$
\begin{aligned}
\mathcal{T}\llbracket\texttt{\#pragma acc data }\textit{data-clauses}\rrbracket &= \texttt{\#pragma omp target data }\mathcal{T}\llbracket\textit{data-clauses}\rrbracket\\
\mathcal{T}\llbracket\texttt{if(}\textit{condition}\texttt{)}\rrbracket &= \texttt{if(}\textit{condition}\texttt{)}\\
\mathcal{T}\llbracket\texttt{copyin(}\textit{vars}\texttt{)}\rrbracket &= \texttt{map(to:}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{present\_or\_copyin(}\textit{vars}\texttt{)}\rrbracket &= \texttt{map(to:}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{copyout(}\textit{vars}\texttt{)}\rrbracket &= \texttt{map(from:}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{present\_or\_copyout(}\textit{vars}\texttt{)}\rrbracket &= \texttt{map(from:}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{copy(}\textit{vars}\texttt{)}\mid\texttt{ present\_or\_copy(}\textit{vars}\texttt{)}\rrbracket &= \texttt{map(tofrom:}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{present(}\textit{vars}\texttt{)}\rrbracket &= \texttt{map(tofrom:}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{create(}\textit{vars}\texttt{)}\rrbracket &= \texttt{map(alloc:}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{present\_or\_create(}\textit{vars}\texttt{)}\rrbracket &= \texttt{map(alloc:}\textit{vars}\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}\llbracket\texttt{\#pragma acc parallel }\textit{par-clauses}\rrbracket &= \texttt{\#pragma omp target teams }\mathcal{T}\llbracket\textit{par-clauses}\rrbracket\\
\mathcal{T}\llbracket\texttt{num\_gangs(}n\texttt{)}\rrbracket &= \texttt{num\_teams(}n\texttt{)}\\
\mathcal{T}\llbracket\texttt{vector\_length(}n\texttt{)}\rrbracket &= \texttt{thread\_limit(}n\texttt{)}\\
\mathcal{T}\llbracket\texttt{private(}\textit{vars}\texttt{)}\rrbracket &= \texttt{private(}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{firstprivate(}\textit{vars}\texttt{)}\rrbracket &= \texttt{firstprivate(}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{reduction(}\textit{vars}\texttt{)}\rrbracket &= \texttt{reduction(}\textit{reductions}\texttt{)}\\
\mathcal{T}\llbracket\texttt{independent}\rrbracket &= \epsilon
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}\llbracket\texttt{\#pragma acc update }\textit{update-clauses}\rrbracket &= \texttt{\#pragma omp target update }\mathcal{T}\llbracket\textit{update-clauses}\rrbracket\\
\mathcal{T}\llbracket\texttt{host(}\textit{vars}\texttt{)}\rrbracket &= \texttt{from(}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{device(}\textit{vars}\texttt{)}\rrbracket &= \texttt{to(}\textit{vars}\texttt{)}\\
\mathcal{T}\llbracket\texttt{if(}\textit{condition}\texttt{)}\rrbracket &= \texttt{if(}\textit{condition}\texttt{)}
\end{aligned}
$$

$$
\mathcal{T}\llbracket\texttt{\#pragma acc loop }\textit{type}\ \textit{loop-clauses}\rrbracket =
\begin{cases}
\texttt{\#pragma omp distribute parallel for }\mathcal{T}\llbracket\textit{loop-clauses}\rrbracket\\
\qquad\text{if } \textit{type} = \texttt{gang vector}\\
\texttt{\#pragma omp distribute }\mathcal{T}\llbracket\textit{loop-clauses}\rrbracket\\
\qquad\text{if } \textit{type} = \texttt{gang}\\
\texttt{\#pragma omp parallel for }\mathcal{T}\llbracket\textit{loop-clauses}\rrbracket\\
\qquad\text{if } \textit{type} = \texttt{vector}\\
\epsilon \qquad\text{if } \textit{type} = \texttt{seq}
\end{cases}
$$

Figure 4.2: Translation rules from OpenACC to OpenMP 4.
The symbol $\epsilon$ denotes the empty string.



```
#pragma acc data copy(data[0:N], result[0:N])
    {
#pragma acc parallel loop if(0)
        for (int i = 0; i < 1; i++) {
#pragma acc loop private(tmp)
            for (int j = 0; j < 5; j++) {
                tmp = 0;
#pragma acc loop reduction(+:tmp)
                for (int k = 0; k <= j; k++) {
                    tmp += data[k];
                }
                result[j] = tmp;
            }
        }
    }

    return 0;
}
```

```
#pragma omp target data map(tofrom:data[0:N], result[0:N])
    {
#pragma omp target teams if(0)
#pragma omp distribute
        for (int i = 0; i < 1; i++) {

            for (int j = 0; j < 5; j++) {
                int tmp;
                tmp = 0;

#pragma omp parallel for reduction(+:tmp)
                for (int k = 0; k <= j; k++) {
                    tmp += data[k];
                }
                result[j] = tmp;
            }
        }
    }

    return 0;
}
```

Figure 4.3: Refactoring Example

# Chapter 5

# Evaluation

To evaluate the correctness of our tool, we used it to translate the EPCC Level 1 OpenACC benchmarks [5], which are ports of the PolyBench and PolyBench/GPU kernels [6]. We also translated several miscellaneous test programs, including naïve and tiled matrix multiplication kernels. We compiled the original OpenACC code using PGCC 16.1 and the resulting OpenMP code using Clang 3.8. We also compiled and measure performance of original OpenACC and refactored OpenMP code using Cray compilers. Performance measurements were taken on a Tesla K40.

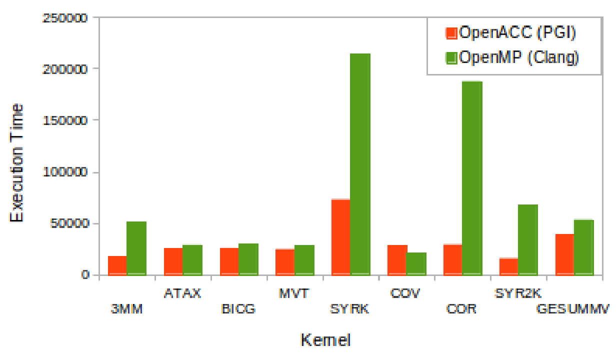
The EPCC Level 1 benchmarks contain a total of 70 OpenACC directives, three of which are *kernels* directives that our tool would not translate directly. Of the remaining 67 directives, 13 were data directives, 23 were parallel loop directives, and 31 were loop directives nested under other parallel directives. There was only one OpenACC API call: a call to `acc_init`, used to initialize the GPU and the OpenACC runtime.

This suggests that a tool like ours could be quite valuable. Only two changes needed to be made by the programmer: removing `acc_init` and converting the three *kernels* regions. The remaining 67 directives could all be converted automatically, without manual intervention.

Interestingly, while both OpenACC and Clang were targeting the same GPU device, the GPU kernels generated by Clang were almost always slower than those generated by PGCC. The naive matrix multiplication kernel in Figure 5.1 can serve as a small, concrete example. With $2048 \times 2048$ matrices, the OpenACC kernel ran for about 250 ms; the OpenMP kernel ran for about 925 ms. Since the OpenACC and OpenMP codes were nearly identical, and they were running on the same hardware, we did not expect such massive differences in

```
#pragma omp target teams distribute
#pragma acc parallel loop gang
for (int j = 0; j < N; j++) {
    #pragma omp parallel for
    #pragma acc loop vector
    for (int i = 0; i < N; i++) {
         double t = 0.0;
        #pragma acc loop seq
        for (int k = 0; k < N; k++) {
            t += m[k][i] * n[j][k];
        }
        p[j][i] = t;
    }
}
```

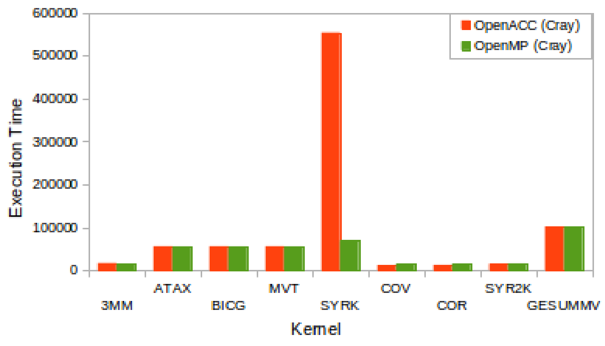
Figure 5.1: Naïve matrix multiplication.



| Kernel | OpenACC (PGI) | OpenMP (Clang) | Speedup |
|---|---|---|---|
| 3MM | 17241 | 50073 | 0.34 |
| ATAX | 24539 | 28308 | 0.87 |
| BICG | 25178 | 29561 | 0.85 |
| MVT | 24021 | 27928 | 0.86 |
| SYRK | 72562 | 213693 | 0.34 |
| COV | 28120 | 19905 | 1.41 |
| COR | 28640 | 187218 | 0.15 |
| SYR2K | 15231 | 66935 | 0.23 |
| GESUMMV | 37998 | 52765 | 0.72 |
| | | *Median:* | *0.72* |

Figure 5.2: Execution times for EPCC Level 1 benchmarks on a Tesla K40 ($\mu s$).

performance. Execution times for the EPCC Level 1 benchmarks on a Tesla K40 are shown in Table 5.2. We ran the same benchmarks on Cray compilers where we have both OpenACC and OpenMP support. In Cray, we got comparable execution times between OpenACC and OpenMP using same code. Execution times for Cray are shown in Figure 5.3. In both cases the benchmarks were run with a data size of 1024 and 5 repetitions. Clearly, details of the compiler-generated code can have a substantial impact on performance.



| **Kernel** | **OpenACC (Cray)** | **OpenMP (Cray)** | **Speedup** |
|---|---|---|---|
| 3MM | 16038 | 13277 | 1.21 |
| ATAX | 55326 | 55101 | 1.00 |
| BICG | 55319 | 55042 | 1.01 |
| MVT | 55374 | 55153 | 1.00 |
| SYRK | 552284 | 68408 | 8.07 |
| COV | 11458 | 14042 | 0.82 |
| COR | 11602 | 14173 | 0.82 |
| SYR2K | 14391 | 15402 | 0.93 |
| GESUMMV | 102011 | 102018 | 0.99 |
|  |  |  | *Median: 1.00* |

Figure 5.3: Execution times for EPCC Level 1 benchmarks on Cray ($\mu s$).

# Chapter 6

# Related Work

Most related to our work is a talk by Hernandez et al. at OpenMPCon 2015 [7], [8]. This talk describes the salient differences between OpenACC 2.0 and OpenMP 4.0 and suggests a (manual) procedure for porting code from OpenACC to OpenMP. Their procedure consists of five steps: removing constructs with no OpenMP counterparts (like kernels), translating data regions, translating data updates, translating accelerator parallel regions, and adjusting function attribute specifiers.

Lee and Vetter [10] evaluate existing directive-based models and show that directive-based models can achieve reasonable performance compared to hand-written GPU codes. For accelerator programming, the two most promising options are OpenACC and OpenMP.

Wienke et al. [14] compare both models with respect to their programmability to assist developers in deciding which approach to take.

Xu and colleagues [16] demonstrate the effectiveness of a hybrid model combined with OpenACC and OpenMP as a plausible solution to port scientific applications in heterogeneous architectures.

# Chapter 7
# Conclusion

Our tool is a prototype, and our translation algorithm supports only the most commonly used OpenACC directives. Some areas for improvement are obvious. The algorithm needs to be expanded to include the entire OpenACC specification, including asynchronous kernel launches, unstructured data lifetimes, and OpenACC API calls. Our tool works only on C, while OpenACC and OpenMP both support C++ and Fortran. While the EPCC benchmarks were useful for evaluating our prototype, we need to test its effectiveness on larger code bases (i.e., application codes).

While automating a tedious 1–1 translation is helpful, it is not the end of the conversion process. A common use case will likely be converting OpenACC code to take advantage of the forthcoming second-generation Intel Xeon Phis. Our translation algorithm (and tool) could certainly be specialized for this process.

After the initial OpenMP code has been created, some restructuring will almost certainly be necessary. For example, we have performed some preliminary experiments with Intel's compilers on first-generation Phis; in one case, we needed to convert global variables to local variables to ensure that the OpenMP *target data* directive would guarantee a copy of the arrays to the Phi. Cataloging (and perhaps automating) such transformations would be beneficial.

Perhaps more importantly, performance portability is not guaranteed. A direct translation of OpenACC code optimized for an NVIDIA GPU will not necessary perform at its peak on a Xeon Phi. As noted above, we did not even obtain comparable performance targeting the *same* hardware.

We have described an algorithm (§4) and a prototype tool that converts a subset of OpenACC (Figure 3.1) to OpenMP 4 [13]. The translation algorithm was designed to be simple and predicatble, since it will necessarily be used in conjunction with manual restructuring and performance tuning. We evaluated our tool by converting the EPCC Level 1 OpenACC benchmarks from OpenACC compiled by PGI C/C++ to OpenMP 4 compiled by Clang. Both targeted an NVIDIA GPU (Tesla K40). To keep the translation process transparent, the programmer was required to manually convert OpenACC *kernels* loops, although the tool was able to automatically translate the remaining 67 of 70 directives.

# Bibliography

[1] The OpenACC application programming interface, version 2.5. `http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf`. Accessed June 15, 2016.

[2] OpenMP application programming interface, version 4.5. `http://www.openmp.org/mp-documents/openmp-4.5.pdf`. Accessed June 15, 2016.

[3] Optimizing CUDA. `http://gpgpu.org/wp/wp-content/uploads/2009/06/04-OptimizingCUDA.pdf`.

[4] J. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, San Francisco, CA, 2002.

[5] EPCC OpenACC benchmark suite. `https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite`. Accessed April 29, 2016.

[6] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.

[7] O. Hernandez, W. Ding, W. Joubert, D. Bernholdt, M. Eisenbach, and C. Kartsaklis. Porting OpenACC 2.0 to OpenMP 4.0: Key similarities and differences. `http://openmpcon.org/wp-content/uploads/openmpcon2015-oscar-hernandez-portingacc.pdf`. Accessed April 29, 2016.

[8] O. Hernandez, W. Ding, W. Joubert, D. Bernholdt, M. Eisenbach, and C. Kartsaklis. YouTube: Porting OpenACC 2.0 to OpenMP 4.0: Key similarities and differences. `https://www.youtube.com/watch?v=CHMrcMUXuuY`. Accessed April 29, 2016.

[9] D. B. Kirk and W.-m. Hwu. *Programming massively parallel processors: a hands-on approach.* Morgan-Kaufmann, 2012.

[10] S. Lee and J. S. Vetter. Early evaluation of directive-based GPU programming models for productive exascale computing. In *Proc. SC12*, page 23. IEEE Computer Society Press, 2012.

[11] C. NVIDIA and C. PGI. The OpenACC specification, version 1.0, 3 november 2011. *Cited on*, page 28.

[12] OpenMP 4.0 on NVIDIA CUDA GPUs. `https://parallel-computing.pro/index.php/9-cuda/43-openmp-4-0-on-nvidia-cuda-gpus`. Accessed April 29, 2016.

[13] N. Sultana, A. Calvert, J. L. Overbey, and G. Arnold. From OpenACC to OpenMP 4 : Toward automatic translation. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, page 44. ACM, 2016.

[14] S. Wienke, C. Terboven, J. C. Beyer, and M. S. Müller. A pattern-based comparison of OpenACC and OpenMP for accelerator computing. In *Euro-Par 2014 Parallel Processing*, pages 812–823. Springer, 2014.

[15] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Boston, MA, 1995.

[16] R. Xu, S. Chandrasekaran, and B. Chapman. Exploring programming multi-GPUs using OpenMP and OpenACC-based hybrid model. In *IPDPSW '13*, pages 1169–1176. IEEE, 2013.