University of Glasgow | School of Computing Science

Domain-Specific Automatic Parallelisation of Scientific Code

Gavin Davidson (1102607d)

22nd April 2016

ABSTRACT

General purpose graphics hardware represents a powerful and affordable tool for climate scientists who look to model the vastly complex systems that make up the Earth's atmosphere. However, leveraging these highly parallel devices in this field requires an understanding of tools like OpenCL and effective strategies for parallelisation. We present a source to source compiler that automatically detects parallelism in sequential, scientific Fortran code and produces programs parallelised using OpenCL. Our approach requires no directives or extra information from the user and output code is composed with use of algorithmic skeletons. We evaluate our work using a version of the large eddy simulator along with synthetic test cases and show that this approach yields performance increases with minimal effort.

1. INTRODUCTION

The behaviours of Earth's atmosphere and weather systems are extremely complex and affected by a great many variables. Software models of these atmospheric conditions are used to simulate and predict meteorological phenomena but are often encumbered by the complexity of the physical system. These complicated programs have extremely long running times, even using modern, high speed processors.

An example of such a program is the *large eddy simulator* (LES) which models turbulence and air currents. Since the proposal of the original Smagorinsky-Lilly model [**smagorinsky1963general**] the LES has been applied to boundary layer flows over urban areas [**nakayama2012large**], combustion in gas turbines [**schmitt2007large**] and aircraft noise modelling [**wang2000computation**] among many other areas. Usually the bulk of the computations performed by large eddy simulators are in the form of repeated calculation on many data points.

In recent years, since CPU clock speed increases have ground to a halt, software developers and scientists alike have looked to more parallel hardware for increased performance. Most commonly this refers to multicore CPUs that are now commonplace in even the most unexceptional computers. However, another type of hardware that has come to the fore in the area of high performance computing is the *graphics processing unit* (GPU). These devices, previously only used to generate visual stimuli, represent a powerful, affordable medium for high performance computing. While the architectures of the individual cores of these devices are less sophisticated than CPU cores, the sheer number of stream cores found on a GPU allows for highly parallel processing. Frameworks such as OpenCL [**OpenCL_home**] and CUDA [**CUDA_home**] allow a programmer to leverage their graphics devices to computationally intensive tasks and have facilitated the growth of general purpose graphics processing (GPGPU). The use of GPUs and parallel programs can result in performance increases of orders of magnitude over sequential versions of the same programs.

The nature of meteorological simulators, in that ordinarily the same calculations are repeated on many data points, make them ideal candidates for performance increases through use of general purpose graphics processing. Scientists who do their work using climate simulations are often equipped with the skill to write programs that simulate their mathematical models. However, writing parallel programs that effectively use GPU hardware is a challenge even for the most experienced programmer and it is unlikely that those who employ LES will have the necessary knowledge to do this.

This paper will describe an automatically parallelising compiler that is tailored to the compilation of programs written in Fortran that carry out meteorological simulations, such as the large eddy simulator. The input source does not need to be changed in any way for use with our compiler, as it would be when using a directive based approach to automatic parallelisation. The process employed by the compiler is one of attempting to recognise patterns in the sequential versions of programs and applying transformations to the source code appropriately. Dependency analysis is performed to be assured that transformations will produce correct output in the logic of the implemented algorithm and errors are reported to a user in the event that a code segment cannot be parallelised for any reason. The final output from the compiler is Fortran code parallelised using OpenCL.

The production of high level source rather than compiled machine code causes this tool to be classified as a *source to source compiler*. This aspect gives more advanced users the ability to perform further transformations on the final output programs such as hardware specific optimisations or even the addition of documentation.

1.1 Statement of Problem

In previous work by Vanderbauwhede [**vanderbauwhede2015mod**] a version of a LES was automatically converted from Fortran-77 to Fortran-95 using a refactoring compiler. The transformed source code was then manually adapted to employ OpenCL to exploit the inherent parallelism of the program. When run on graphics hardware, the adapted program demonstrated a 7x speed up in run time when compared to the original version of the program run on a CPU. This result shows that there is an opportunity for performance gains in this type of simulator and the paper itself states a need for a tool that automatically parallelises programs like the LES.

Such a tool would allow climate scientists to leverage powerful, affordable graphics hardware when performing such computationally intensive simulations as the LES. Using an automatically parallelising compiler would mean that those interested in weather simulation need not have knowledge of complex parallel programming frameworks like OpenCL and also that the time taken to write such complicated programs is eliminated. The general result would be higher productivity for these scientists.

1.2 Aims

It was the aim of the current work to produce a tool that analyses sequential Fortran programs and produces parallel Fortran programs structured as OpenCL host and kernel code. It is important to note however that we are not attempting to create a general purpose tool with this project, rather a domain specific compiler that therefore affords us the ability to focus on patterns that occur often in scientific Fortran programs. As such, the problem is greatly reduced and is much more feasible while still being a useful and interesting prospect. However, the success of this project is still considered a stride toward a more general purpose, automatically parallelising compiler.

2. BACKGROUND

The invention of an automatically parallelising compiler has been a long standing goal in the field of high performance computing. The creation of such a tool that could be applied to any sequential program to effectively take advantage of any available parallelism would constitute a huge leap forward in compiler technology. This section will present some of the abundant amount of work that has been done in pursuit of automatic parallelisation. This will include an exploration of dependency analysis, algorithmic skeletons and source to source compilation. There will also be a discussion on general purpose graphics programming.

2.1 Dependency Analysis

Dependency analysis is a technique used in code analysis and transformation to discover how different parts of a program rely on one another. For example, it is useful to know whether a set of statements must be executed in the order that they appear or whether they can be rearranged and run in a different order. In the context of automatic parallelisation, dependency analysis is used to determine whether the execution of an iteration of a chosen loop is independent of the previous and following iterations. In such a case, it may be possible to execute all iterations of the loop simultaneously, in parallel.

Austin and Sohi [**austin1992dynamic**] describe a technique for dependency analysis involving a *Dynamic Dependency Graph* and use it to automatically parallelise sequential programs. However, since this work pre-dates the widespread use of massively parallel processor devices and frameworks such as OpenCL and CUDA, the authors attempt to take advantage of *task parallelism* rather than *data parallelism*. Essentially this means that their solution attempts to determine whether program segments performing different tasks can be run at the same time on a parallel system of some sort. This contradicts modern approaches in high performance computing that focus more on performing the same task on many pieces of data simultaneously. It is the more modern approach of data parallelism that is employed by the Fortran compiler produced by this project.

Ferrante et al. [**ferrante1987program**] also present a graph based approach to dependency analysis. Their *Program Dependency Graph* (PDG) contains information on both *data dependence* and *control dependence*. The term data dependence refers to the situation where the value of one variable is calculated directly using another variable, upon which it therefore depends. This form of dependency is the focus of Austin and Sohi's work [**austin1992dynamic**]. Conversely, control dependence occurs when the value of a variable or array element is determined by some form of conditional branch in a program. In this case, the variable or array depends on the predicate that determines whether or not the conditional branch is executed. The work by Ferrante et al. [**ferrante1987program**] goes on to explain applications of the PDG which include detecting parallelism and situations where loop fusion is possible. The conditions for loop fusion that are presented here form the basis of the conditions employed by our work. Our conditions, however, are more relaxed and allow a larger degree of flexibility when it comes to deciding whether two loops can be fused (see section **??**). The rules set out by Ferrante et al. for detecting parallelism contribute, to a lesser extent, to the formulation of rules for parallelism in our compiler.

A strategy that does not involve building graphs for dependency analysis is employed by Oancea and Rauchwerger [**oancea2012logical**]. Their compiler framework performs both static and dynamic dependency analysis. The terms *static* and *dynamic* dependency analysis used here mean that analysis is performed before and during run time, respectively. Analysis is performed using *uniform set representation* - a language that is designed to be composable and able to describe dependencies. The aim of this compiler framework is loop parallelisation and so the collected dependencies are used to determine whether a loop's iterations are independent. A loop iteration is independent if the order in which all of the iterations of the loop are executed has no bearing on the overall result of the program. Oancea and Rauchwerger's work defines a set of sufficient conditions for a particular loop to be parallelisable and attempts to resolve them before and during runtime. While the use of dynamic analysis and sufficient conditions for parallelism separates this compiler framework from our Fortran compiler, the evaluation strategy is similar (as is Austin and Sohi's [**austin1992dynamic**], if outmoded). Oancea and Rauchwerger evaluate the success of their work by parallelising sequential programs, running the new versions on a multicore CPU and comparing the results to the sequential version run on the same CPU. The evaluation of our Fortran compiler takes inspiration from this and involved testing that the new parallel versions of code did indeed run successfully on parallel devices like GPUs.

2.2 Algorithmic Skeletons

Algorithmic skeletons are frameworks of common patterns and sequences of operations that can be used to easily introduce parallelism to sequential code with little knowledge of parallel hardware. In many cases, a programmer is provided with a set of functions and data types that allow them to apply common parallel patterns to their code. This approach, which requires that the programmer has some understanding of parallelism but hides most of the complexity, will be explored in this section. Another approach is identifying pat-

terns in sequential programs automatically and producing appropriate parallel code that makes effective use of whatever hardware is to be used. The second strategy will be discussed in section **??** and is the method that is employed by our Fortran compiler.

SkelCL is a C++ library that intends to eliminate some of the complexity of OpenCL by removing the necessity of a large amount of boilerplate code. The library, presented by Steuwer et al. [**steuwer2011skelcl**], uses algorithmic skeletons in conjunction with a new abstract vector data type to give a programmer a more high level view of the parallelism made available by OpenCL. Another focus of the work is on optimising memory usage and this is done by removing the responsibility of the programmer to deal with transferring data between host and device. While decreasing programming complexity, SkelCL imposes a performance overhead when compared to versions of programs written manually using OpenCL. This is not ideal as the major purpose of parallel processing is performance. Two of the patterns supported by SkelCL are implemented as part of our Fortran compiler - namely *map* and *reduce*. However, these patterns are recognised in source code rather than made available for a programmer to apply.

The use of an abstract vector data type is an idea also adopted by Enmyren and Kessler [**enmyren2010skepu**] in their work into parallel algorithmic skeletons. They describe SkePU, another C++ library for algorithmic skeletons, that produces more than just OpenCL code but also supports the production of CUDA, OpenMP and sequential CPU code. The work labels the operations performed by their abstract vector data type as a *lazy copying technique*, by which they mean memory transfers are only performed when absolutely necessary, in order to reduce overheads. The work supports some interesting variants on the common map and reduce patterns, one of which is supported by our compiler. In particular, the *map reduce* pattern where a function is applied to all elements of a dataset before it is reduced using some other function is recognised by our Fortran compiler.

Marques et al. [**marques2013algorithmic**] present Marrow, another algorithmic skeleton framework. The aim of the work is to introduce a set of skeletons that have not previously been available for GPU development. Where other frameworks have provided variants of the *map* skeleton, Marrow also presents *pipeline*, *stream* and *loop* skeletons. The framework also attempts to take advantage of a concept made available by OpenCL known as *overlap* which essentially attempts to ensure that the GPU is never left idle in order to increase overall performance. The *overlap* functionality adds to the overall aim of algorithmic skeletons which is to lower the complexity of writing parallel programs. During the evaluation of this work, it was found that Marrow introduces a small performance overhead when compared with handwritten OpenCL programs, much like SkeCL [**steuwer2011skelcl**].

2.3 Source to Source Compilation

Traditionally, a compiler is a piece of software that translates a relatively high level programming language into a lower level one. Often the target language is some form of machine code that can be directly interpreted by hardware. However, source to source compilation involves translating a high level language into another language that is of a similar level of abstraction. For source to source compilers, this can sometimes mean translating from one language to the same language, after performing some form of transformation. In the context of automatic parallelisation, a source to source compiler would be used to transform the original source by rearranging it and adding constructs that support parallelism before outputing the new version of the source in a high level language. This new version of the program could then be compiled further into machine level code and run. Our compiler uses source to source compilation in exactly this way, where the input and output languages are both Fortran.

Bones is a source to source compiler, presented by Nugteren and Corporaal [**nugteren2012introducing**], that uses algorithmic skeletons to translate sequential C programs to parallel CUDA and OpenCL code. The compiler applies a set of skeletons according to *pragmas* (or compiler directives) that are placed in the sequential input program. Input source code is parsed and used to construct an *abstract syntax tree* (AST) which includes these pragmas. The compiler then traverses the AST and, where it finds pragmas, performs analysis and applies transformations appropriately. The stated motivation for such a system is that programmers are still willing to put in extra effort to write native OpenCL and CUDA programs if it will maximise performance, even in the presence of tools that make the task easier. As such, the use of algorithmic skeleton frameworks like SkePU [**enmyren2010skepu**] and Marrow [**marques2013algorithm**] suit neither novice programmers, as there is still a level of complexity, or expert programmers, because performance gains are not high enough. Bones attempts to remedy this by presenting skeletons that are more specific to certain scenarios and therefore produce better performance. Nugteren and Corporaal attempt to strike a balance by not making the skeletons so specific that they are hard to use. A further benefit for those interested more in performance is that the output of the compiler is high level and can therefore be optimised more if need be. The approach taken by Nugteren and Corporaal is similar to the one we took with our Fortran compiler. The main difference is the use of pragmas - where Bones requires a programmer to enter pragmas at appropriate places to allow for parallelism, our compiler recognises it automatically, without a need for a user to point it out.

Bae et al. [**bae2013cetus**] present an overview and evaluation of Cetus, a source to source compiler framework that had an original aim of use in automatic parallelisation of C programs. Cetus performs a plethora of analyses and optimisations but the elements that are most associated to this project are the exploration of loop parallelism, reduction detection and data dependency analysis. The rules for detecting reductions that Cetus employs inspired the rules that are used by our Fortran compiler. Bae et al. evaluate Cetus by using it to automatically produce parallel OpenMP code from sequential C. During this evaluation, it was found that Cetus parallelised the sequential input more effectively than a competitor and was only outperformed by handwritten OpenMP code. This success shows the benefit for climate scientists that could be gained from such a compiler being written for Fortran. An added point of interest is that Bae et al. attribute the fact that handwritten code outperformed Cetus's output to the fact that a human can decide whether or not parallelising a certain code block will produce a performance increase due to parallelism or a performance decrease because of the overhead of setting up the parallelism.

The source to source aspect of our compiler allows a programmer to make the final decision as to whether or not a loop should be parallelised.

Cetus is used by Yang et al. [**yang2010gpgpu**] as a basis for their source to source compiler. In this case, however, the compiler does not produce parallel code from sequential code but rather produces optimised code from already parallel code. As motivation for their work, the authors reference how complicated the process of writing GPU code can be but also a desire to not completely remove the programmer from the process. The benefit of their compiler is that a programmer can write their parallel algorithm without having to worry about the time consuming process of applying intricate, hardware specific optimisations. The evaluation showed that the automatically optimised code outperformed even a handwritten and optimised library for the same calculations. The work serves to show that device specific optimisation makes a significant difference to performance. While it is intended that our compiler be as portable as possible, accommodating these device specific optimisations is possible by virtue of the source to source nature of the compiler. Should a more advanced user wish to apply some device specific optimisation, they can very easily by editing the output source code.

A similar undertaking to our own is taken by Lionetti et al. [**lionetti2010source**] in their work on an automatically parallelising source to source compiler. The paper describes a domain specific compiler for Python programs that perform electrophysiological simulations. The intention is the same as our own—allowing scientists to leverage parallel GPU hardware to run their computations without having to write the code themselves. Other than the input and output languages targeted (Python to CUDA vs Fortran to OpenCL), the work also differs from our own in their use of pragmas in the input source code. This means that a user must have at least some knowledge of parallelism to effectively use the system. Employing the same approach as our own, Lionetti et al. [**lionetti2010source**] construct an abstract syntax tree and analyse it for dependencies before CUDA code is finally emitted.

2.4 General Purpose Graphics Programming

The concept of parallel processing is by no means new in computing science and has been investigated for a long time. In the past, the focus of such research has been on *task parallelism* which is the idea of running distinct tasks of the same program at the same time on multiple devices or multiple cores of a device. However, more recently focus has shifted to *data parallelism* as devices with increasing numbers of cores become available. Data parallelism refers to a situation where instances of the same task are performed at the same time on distinct data points. This form of parallelism is perfectly suited to a many-core device like a GPU that has less sophisticated processor cores than a standard CPU but has a great many of them. The performance of a graphics device comes from sheer strength of numbers and the use of these highly parallel systems for non-graphical tasks is facilitated by frameworks like nVidia's CUDA [**CUDA_home**] and Khronos's OpenCL [**OpenCL_home**].

While not the true focus of their work, many of the papers mentioned earlier in section **??** show that the use of GPU hardware allows for large speed ups when compared to sequential code. These works include Nugteren and Corporaal [**nugteren2012introducing**], Lionetti et al. [**lionetti2010source**] and Enmyren and Kessler [**enmyren2010skepu**] and are just a tiny subset of the work that supports the use of GPU hardware for general purpose parallelism.

Graphics devices are not the only choice where efficient parallelism is concerned. Depending on the characteristics of the algorithm that has been implemented, it may be more fruitful to use a multicore CPU or FPGA (field programmable gate array). Asano et al. [**asano2009performance**] make performance comparisons between GPU, CPU and FPGA hardware for some image processing applications in order to try and determine the characteristics for when each type of hardware should be used. In conditions where shared memory usage is high or complex operations are employed, the graphics device is surpassed by the other two devices. However, GPU hardware outperforms CPUs and FPGAs when there is minimal use of shared memory and operations performed by the program are simple. This result supports the use of graphics hardware for parallelising programs like large eddy simulators as these programs perform simple mathematical operations in parallel on large datasets.

A major choice to be made when writing parallel GPU code is which framework to employ. Some major competitors in this area are OpenCL [**OpenCL_home**] and CUDA [**CUDA_home**]. Both frameworks present a similar abstraction of the hardware they allow access to. This abstraction is one of *host* and *device*. A programmer writes one or more *kernels* that are run directly by the GPU (or other device) and a *host program* that is run on the system's main processor. The host program handles memory transfers to the device and initialising computations and the kernels do the bulk of the processing, in parallel on the device. The main difference between the two frameworks is compatibility. CUDA is a proprietary system owned and maintained by nVidia and as such, CUDA only supports nVidia graphics devices. In contrast, OpenCL is much more widely supported. A programmer can parallelise their algorithm using OpenCL and run it on a CPU, FPGA or GPU with minimal changes. There is no manufacturer specific compatibility associated with OpenCL which means that OpenCL can also be used with nVidia devices. This portability is one of the major benefits of OpenCL over CUDA. However, there is much work that shows that CUDA produces better performance than OpenCL when their run times are directly compared [**steuwer2011skelcl**, **enmyren2010skepu**].

A third choice for parallel programming is OpenACC which is a directive based approach that affords a higher level of abstraction for parallel programming than CUDA and OpenCL. In a basic example, a programmer adds *pragmas* (compiler directives) to their sequential code to indicate which parts of the code are to be parallelised. The new source code, including directives, is then processed by the OpenACC compiler and programs that can run on parallel hardware like GPUs are produced. There are a number of extra directives that allow for optimisation and tuning to allow for the best possible performance. Herdman et al. [**herdman2012accelerating**] compare OpenACC, CUDA and OpenCL implementations of a hydrodynamics program for performance, productivity and portability. Their metric for productivity involves recording the number of words that had to be added to the sequential versions of the program to make it compatible with each of the frameworks. They found that OpenACC required least changes, and was there-

fore most productive, while also producing the best performance of the three frameworks. OpenCL was found to be most portable.

Other works show that OpenACC is usually slower than CUDA and OpenCL but can have comparable performance when tuned [**wienke2012openacc**, **hoshino2013cuda**]. Our compiler allows an even higher level of abstraction from the device than that allowed by OpenACC as a programmer need not even consider that their program will become parallel. In our approach, a programmer does not need to add anything to their sequential code to identify parallelism or anything else as this is all handled automatically by the compiler.

An attempt at a comprehensive performance comparison between OpenCL and CUDA is performed by Fang et al. [**fang2011comprehensive**]. The work involves testing a suite of algorithms, implemented in both OpenCL and CUDA, on a number of nVidia graphics devices. This test suite includes both *real world* algorithms and algorithms that are simply designed to benchmark the components of the system in use. The authors found that OpenCL and CUDA had the same potential peak performance on the given hardware when running the benchmark algorithms. However, CUDA outperformed OpenCL in most of the real world applications. The authors go some way to explain this and the inconsistency is attributed to differences in the programming model, compiler optimations, optimisations on the kernel code and optimisations for architecture that CUDA performs automatically. When these advantages are equalised, OpenCL is found to have comparable performance with CUDA. The work goes on to explore the portability of OpenCL by running the test suite on a much larger range of hardware of different types and manufacturers.

Since the aim of our Fortran compiler was to increase accessibility of parallelism for meteorological scientists, OpenCL was chosen because of it's portability. The work by Fang et al. [**fang2011comprehensive**] shows that this does not necessarily equate to a performance drop when compared with CUDA and this further strengthens the case for OpenCL in this context.

3. COMPILER ARCHITECTURE

The product of this research is an automatically parallelising compiler that targets sequential Fortran as input and produces Fortran parallelised with OpenCL as output. The compiler itself is written using Haskell and makes heavy use of the *'Data.Generics'* library and the *'scrap your boilerplate'* [**lammel2005scrap**] style of programming. This work forms a basis of the argument for the implementation of an OpenCL API for Fortran as there exists a very large codebase of computationally intensive Fortran programs that could benefit greatly from the use of tools like our compiler in conjunction with OpenCL.

This section details the components of our Fortran compiler and will go into some depth on the conditions for parallelism and loop fusion that it uses. The source code for the compiler can be found in a public github repository[1].

3.1 Parsing

Fortran is an extremely mature and diverse language that supports many versions and variants of standard syntax. As such, the development of a complete parser is a huge task and one that could not be undertaken in the time scale of this project. Instead, our compiler makes use of an open source, third party Fortran parser called *Language-Fortran*. The creation of a preprocessor and a number of modifications to Language-Fortran itself were necessary for it to be suitable for the needs of the project.

Preprocessor

At the beginning of the project, the use of Language-Fortran represented a risk. This was due to the fact that the parser was by no means complete or bug free, but it was decided that we would attempt to avoid any simple bugs that were encountered and adapt functionality as necessary. The bugs in Language-Fortran were minor differences between the syntax it expected and the syntax supported by the standard Fortran compiler. In many cases this was expecting white space before a particular token or not wholly supporting the case insensitive nature of the language. These small issues are handled by a small preprocessor that essentially performs global search and replace operations on the input source.

Language-Fortran

As mentioned previously, Language-Fortran is an open source Fortran parser written by Jason Dagit. The license on the software allows modification under certain conditions that have been met by this project. The source code is available from a public github repository[2]. Modifications to the original version of Language-Fortran were necessary and so the version of the software in the original repository does not match the code that is used by our compiler. However, the adapted version of the parser is provided with the rest of the compiler's files in our github repository.

Language-Fortran's parser component is generated from a grammar specification using the *Happy* parser generator[3]. The grammar file defines a set of production rules, non-terminal symbols and terminal symbols. The grammar specification allows the generated parser to recognise syntax patterns in the input source code and generate an appropriate abstract syntax tree (AST) node.

The input to Language-Fortran's parser is not strictly the source code but is in fact a token stream. This token stream is produced by a lexer that receives the input source (already preprocessed) and attempts to tokenise it. The creation of this lexer is also handled by the Happy parser generator. In this case a much smaller specification is required than that of the parser grammar.

The other major component of the parser infrastructure is a file named *Fortran.hs* that contains specifications for all of the Haskell types that make up nodes of the AST. Almost all of these types, from the top level *Program* type to the frequently used *Expr* type, are adorned with a polymorphic type intended to hold annotations.

The additions and changes made to Language-Fortran were mostly minor. Support for *select case* statements was added with a new production rule in the parser's grammar file and a new type constructor in Fortran.hs. A bug that caused

[1]Compiler (AutoParallel-Fortran) github repository: `https://github.com/KombuchaShroomz/AutoParallel-Fortran`

[2]Parser (Language-Fortran) github repository: `https://github.com/dagit/language-Fortran`

[3]Happy parser generator homepage: `https://www.haskell.org/happy/`

the parser to throw an error when the name of a subroutine was not exactly the same case between its opening and closing statements was fixed with a small modification to the grammar file.

A more substantial update was in changing the annotation type used by the parser. By default, the parser assigns an empty tuple to the polymorphic annotation type provided by Fortran.hs. Instead, the parser now instantiates an empty map data structure that is later populated with *String* keys and *String list* values. The intention here was to use the annotation type to hold information on parallelisation errors (further explained in section **??**).

Finally, type constructors that represented the patterns that the compiler would recognise were added to Fortran.hs. Namely, these patterns are *map* and *reduce* and while they would never be used during the parsing process, these constructors would allow for these patterns to be represented in the program's AST upon detection. This also made code emission a more natural process as the code emitter simply needed to traverse an AST, rather than deal with some extra data structure somewhere.

3.2 Parallelism Detection

The main focus of our research was the detection of parallelism in sequential programs. We sought to recognise specific patterns in the source code and make no claim that our work has produced a general purpose automatically parallelising compiler. The analysis process employed by the compiler involves repeated traversals of an AST produced by our slightly modified Language-Fortran parser. The general strategy for parallelism detection was to detect whether each and every loop in the input file could be rewritten as a parallel OpenCL kernel. As such, most of the analysis deals with a single loop at a time starting with the most deeply nested in the case of nested loops. The process of traversing the AST would have been much more complex and cumbersome if not for the generics provided by the *'Data.Generics'* library.

Dependency Analysis

The dependency analysis performed by our compiler could be considered rather simple when compared to previous work in the field (see section **??**). Essentially, the product of dependency analysis is a table linking variables to the expressions that they are *directly dependent* upon within a particular scope in the source program. This table, implemented as a map of variable names to lists of expressions, is produced by traversing the AST node for the particular code segment and detecting assignment statements. A variable is directly dependent on an expression if the variable appears on the left hand side of the assignment and the expression appears as an operand on the right hand side of the assignment statement.

Since dependency is a transitive property, it was necessary to implement functionality that allowed for the retrieval of expressions that are *indirectly dependent* on a particular variable. A variable is indirectly dependent on an expression if it is possible to trace a path through the dependency table from that variable to the expression via other variable to expression dependencies. Consider the following example:

```
C = D - 22
B = C * 14
A = B + 1
```

In this example, A is directly dependent on B and indirectly dependent on C and D.

During the detection process for parallel maps and reductions, the concept of *loop carried dependency* is utilised. A function that checks whether a loop exhibits a loop carried dependency is defined as part of our analysis and is implemented using a definition based on that of Allen and Kennedy [**allen2002optimizing**]. Loop carried dependency occurs because of the iteration of loops. If a loop's body contains two statements that, in different iterations, point to the same place in memory then there is a loop carried dependency.

More formally, given two statements **P** and **Q** that access the same array in a single loop, values of the loop iterator **i** and **j** and a memory location **M**, a loop carried dependency exists under the following conditions:

1. When the loop iterator is **i**, statement **P** references location **M** in memory.
2. When the loop iterator is **j**, statement **Q** references location **M** in memory.
3. **i** and **j** are not equal.

A simplification of this rule is used by our compiler. Rather than evaluate all of the addresses that are accessed during the complete run of a loop, for each array assignment the analysis first extracts all other assignments to that array from the loop. Those accesses, including the one being analysed, are then filtered down to only those that use the loop iterator as part of the index/offset expression. The check for that assignment passes if all of the index access expressions (that use the loop iterator) are the same. That is to say, given a value of the loop iterator, only one index in the array in question will be accessed in that iteration of the loop.

Access Analysis

Access analysis provides our compiler with a number of useful abilities. Data gathered during this process allows for the identification of temporary and non-temporary variables, function calls and values for variables at certain points.

The simplest aspect of this stage of analysis is the extraction of declared variable names and arguments to the current program block. Since the syntax for array accesses and function calls in Fortran can be identical, it is necessary to use the declared variables and arguments to determine whether a statement performs a function call or an array access. The test is simple: if the item being accessed or called is declared at the start of the program or appears in the arguments then this statement is an array access. Otherwise, it is a function call.

More complex analysis is required to identify temporary variables and values for variables at certain program blocks. This process involves traversing the entire program to collect a list containing the location of every read, location of every write and every expression assigned to every variable. In this context, location refers to the location within the input source and is characterised by a line number.

Determining whether or not a variable is temporary or non-temporary in a loop is useful during parallelism detection because if a variable can be proved to be temporary then the conditions that the variable must fulfil to allow

for parallel processing are relaxed. This information can be gleaned from the read and write locations gathered from the whole program.

A variable **V** can be deemed temporary within a loop if either of the following conditions are met:

1. Variable **V** is never read again after the end of the loop.

2. Variable **V** is written to before it is read again after the end of the loop.

Being able to produce an expression for the value of a variable at a certain location in the source in useful when a parallel reduction has been detected. When transforming the abstract syntax tree to include an *OpenCLReduce* kernel, it is necessary to include assignments to *reduction variables* (explained later in this section) that set them to initial values for the reduction. This helps to ensure correctness when the reduction kernel is run in parallel. The expression for a variable at a certain location is produced by finding the assignment to that variable that happened most recently before the target location, using the data constructed during the variable access analysis process.

Map Detection

Of the two patterns that are detected by our compiler, the map is the simplest and usually most effective when run on GPU hardware. In the purest form, a map happens when a number of data points are processed individually, with no interaction with other points. For example, a program that takes a numerical dataset and multiplies every value by four can be considered a map. This structure fits the model of execution of OpenCL perfectly.

The parallelism detection process involves treating each loop in the input source code individually, beginning with the most deeply nested loop in a cluster. However, before the compiler begins to examine each loop, the whole program is analysed using the *access analysis* techniques explained previously. From there, each loop in turn is considered. The first step for each loop is to perform dependency analysis and use that analysis to produce a list of non-temporary variables. A list of variables that are written to during the loop is also compiled. These data structures, along with the variable access analysis data, are then used to determine whether the following conditions are met by the loop:

1. Any access to a non-temporary variable that is written to during an iteration of the loop must make use of all of the loop iterator variables as indexers. For example, a loop that iterates over **i**,**j** and **k** must only ever access any non-temporary, written to array at position **i** + a, **j** + b, **k** + c where a, b and c are arbitrary offset values.

2. Any write to a variable must not form a loop carried dependency (defined in section **??**) on that variable.

3. Any function call must appear as part of an assignment statement, and nowhere else.

Provided these conditions are met, the current loop can be considered a map and an *OpenCLMap* node is added to the abstract syntax tree.

Reduction Detection

The reduction pattern is another extremely common structure in large data processing programs. While a method for effective reductions using parallel hardware may not be as immediately obvious as the map pattern, powerful reductions can be performed with use of some of the more intricate details and features of OpenCL. A reduction, in its truest form, involves some form of aggregation of a large number of data points into a single data point. For example, finding the average or sum of a large number of numerical values is a reduction.

During parallelism detection, if a loop is found to violate one of the conditions for it to be a map then checks are performed to determine whether the loop can be considered a reduction. As before, non-temporary variables, dependencies for the loop, variable access analysis for the program and a list of variables that are written to in the loop are used during an analysis process that checks whether the following conditions are met:

1. Any access to a non-temporary variable that is written to during an iteration of the loop must make use off all of the loop iterator variables as indexers. Otherwise, the variable in question must be a **reduction variable**.

2. Any write to a variable must not form a loop carried dependency (defined in section **??**) on that variable.

3. Any function call must appear as part of an assignment statement, and nowhere else.

The conditions for a variable to be a reduction variable are as follows:

1. Any access to the variable in question must not use all of the loop iterator variables as indexers.

2. The variable in question must, at some point in the loop, be assigned a value related to itself. However, the variable in question must only appear once on the right hand side of the assignment operation, including variables that are in some way dependent on the possible reduction variable.

3. When the variable in question is assigned a value related to itself, it must be done so using an associative function as the primary operator on the variable. That is to say, any operation that is performed directly on the reduction variable must be associative.

These conditions allow for the loop to be performed in parallel as a reduction where the order in which data points are processed doesn't matter. Should they be met, the abstract syntax tree has an *OpenCLReduce* node added to it.

3.3 AST Transformation

Once opportunities for parallel maps or reductions have been identified, the abstract syntax tree of the input program is transformed. Most of the changes involve the removal of loop constructs and the addition of *OpenCLMap* and *OpenCLReduce* nodes. As a way to note which parts of the AST have been changed, any node that has been transformed loses its original source line information which is instead replaced with a node that represents the fact that this

part of the AST has been generated. The addition of these generated source information nodes allows code generation to be much simpler than it would be otherwise.

An important thing to note is that because parallelism detection begins with the most nested loop in a set, a nested loop that can be parallelised will result in a set of nested OpenCL kernel nodes in the AST. This nesting is resolved during the next stage of compilation and is explained in section **??**.

OpenCLMap Kernels

Other than the standard nodes attached to every node of the AST (annotations and source code line information), the OpenCLMap node contains four children:

- **Read variables**: During map detection, a list of variables that a particular loop reads is compiled. This is necessary for the emission of the OpenCL kernels as access modifiers are applied to kernel arguments.
- **Written variables**: In the same way as for the read variables mentioned previously, a list of variables that are written to in the body of a loop that is being parallelised is compiled during the detection processes.
- **Loop iterators**: Initially, when a map is detected, only one loop iterator specification will be stored here. However, when kernels are combined (explained later in this section), this node will house many loop iterator specifications. These specifications themselves contain a variable name for the loop iterator, its starting value, ending value and the increment that is added after each iteration.
- **Body of code**: Contained here is a *Fortran* node that describes the body of the loop that had been parallelised. Essentially, this node holds all of the functionality of the kernel that will be produced from this particular part of the AST.

OpenCLReduce Kernels

OpenCLReduce nodes are structured in exactly the same way as OpenCLMap nodes, other than the addition of one child node. The structure of this node is as follows:

- **OpenCLMap children**: All of the children of OpenCLMap nodes are included as part of OpenCLReduce nodes. Namely, read variables, written variables, loop iterators and code body. These children are used in the same way.
- **Reduction variables**: Reduction variables are specific to a reduction kernel. As mentioned in section **??**, reduction variables are the variables in which the results of the reduction are stored. For more efficient use of the target parallel device, it is necessary to treat statements involving these variables differently during code emission. Therefore, these variables are gathered during the analysis process and stored as a list on the OpenCLReduce nodes.

3.4 Loop Fusion

Loop fusion is a strategy employed by compilers to optimise input programs. The concept is that if two loops iterate over the same range of values and there are no dependencies between the two loops, then the operations of both loops can be performed in a single loop.

The result of loop fusion is clearly a lower number of loops in a program with the same functionality. Since our compiler attempts to translate loops into parallel OpenCL kernels, a further result of loop fusion by our compiler is less kernels. This is an optimisation for our system because the process of starting a kernel in OpenCL has a certain performance overhead and reducing the number of kernels that must be started during the runtime of a program reduces the appearance of this overhead.

The process of loop fusion is carried out after the detection of parallelism in the source code. That is to say, our loop fusion techniques do not actually fuse loops but rather they fuse OpenCL kernel nodes, that were once loops, in an AST. The reason for this is due to the output from the AST transformation process in that nested loops in the source code are transformed into nested kernels in the AST. These nested kernels must be resolved into singular, flat kernels before code emission and it was therefore logical to include the traditional loop fusion procedure at this stage of the compilation processes.

Nested Loop Fusion

The process of fusing nested kernels is simple compared to that of fusing adjacent kernels. Given kernel **X** and kernel **Y**, where **Y** is nested in **X**, the kernels may be fused if the following condition is satisfied:

1. Kernel **X**'s code body contains only kernel **Y** and no other statement.

If it is found that a pair of kernels may be fused under this condition, the node representing the outer kernel is replaced with a new node representing the fused kernel. This condition means that only two kernels of the same type that are nested will be fused because the decision process during parallelism detection will be exactly the same for the inner and outer kernel. The children of the new kernel nodes are produced as follows:

- **Reads**: Since the outer kernel contained only the inner kernel and no other statements, the read variables are the read variables of the inner kernel.
- **Writes**: For the same reason, the written variables are the written variables of the inner kernel.
- **Loop iterators**: The list of loop iterators is produced by concatenating the lists of loop iterators from both the outer and inner kernel. It is important, for code emission later, that the loop iterators of the outer kernel appear first in the new list.
- **Code body**: The body of the new kernel is the body of the inner kernel, again because the outer kernel only contained the inner kernel.
- **Reduction variables**: In the case of fusing two OpenCLReduce kernels, the list of reduction variables is simply the concatenation of the reduction variable lists of the two kernels.

The process of fusing nested kernels begins by trying to fuse to most nested kernels until eventually a singular, flat kernel is left.

Adjacent Loop Fusion

The process of fusing adjacent kernels is closer to the more traditional sense of loop fusion than the nested loop fusion explained earlier. In this context, a loop or kernel is adjacent to another loop or kernel if they appear in the source code directly before and after one anther, with no other statements appearing between them. The conditions employed by our compiler are based on those laid out by Ferrante et al. [**ferrante1987program**] but are somewhat more relaxed to allow for more opportunities to apply this optimisation.

Given kernel **X** and kernel **Y**, where **X** and **Y** are adjacent, and a numerical bound **B**, the two kernels may be fused if the following conditions are met:

1. The loop iterator variables of **X** and **Y** have the same name.
2. The loop iterators of **X** and **Y** have the same start and increment values.
3. The end values of the loop iterators of **X** and **Y** have a difference less than bound **B**.
4. The newly formed kernel does not exhibit a *loop carried dependency* (explained in section **??**).

If these conditions are met, both kernel nodes are removed from the AST and replaced with a new node representing their fusion. The children of the fused node are constructed as follows:

- **Reads**: The read variables of the new kernel is a list containing all of the read variables from both of the original kernels.
- **Writes**: As with the read variables, the written variables are a concatenation of the written variables from the original kernels.
- **Loop iterators**: In the most simple case, two non-nested kernels, the kernel iterator list contains the iterator specification that produces the largest number of iterations of the two kernels' iterator specifications. In a more complex case, two nested kernels, the kernel iterator list contains the instances of each of the kernel iterators that produces the largest number of iterations. For example, the combined kernel iterators for two nested kernels that both iterate over **x** and **y** would be the version of the **x** iterator that produces the largest range of values for **x** and the **y** iterator that produced the largest number of values for **y**.
- **Code body**: When combining kernels that have exactly the same loop iterators, the code body simply contains the statements from the earliest occurring kernel followed by the statements from the second kernel. However, in the case that the loop iterators are different, one of the sets of statements from either of the original kernels must be wrapped in a conditional statement. This conditional statement ensures that the code from the kernel with the smaller number of iterations is executed the correct number of times.
- **Reduction variables**: In the case of two OpenCLReduce kernels, the reduction variable list is the concatenation of the reduction variable lists from the two original kernels.

3.5 Code Emission

The concluding stage of the compilation process is to produce the final output Fortran code. This is done by traversing the newly transformed abstract syntax tree and firstly producing host code. Latterly, the new OpenCLMap and OpenCLReduce nodes are separated from the AST and used to synthesise kernel code.

The process of code generation is made simpler by virtue of keeping track of which parts of the AST have been transformed and which have not. As mentioned previously, those nodes that have been changed during the compilation processed are tagged with source line information that represents *generated* nodes. It is therefore possible to produce a large part of the host code and, to a lesser extent, kernel code by simply copying the lines from the input file that have not changed. This means that code synthesis functionality need only be concerned with code that is produced by the transformation processes, rather than the whole Fortran specification.

The focus of the compiler is on identifying parallelism and therefore the focus at this stage was more on the production of correct parallel kernels rather than the boilerplate code necessary for the host program.

Host Code Emission

The host code produced by our compiler could be viewed as the original source code with all of the parallel loops removed and placed in a different file. The host code file has the same structure as the original source along with all of the same imports, declarations and incoming arguments. In the place of loops that have been parallelised, there are now calls to these new external kernels. Since the output is Fortran, the kernels are implemented as subroutines in their own file.

Where a call to a map subroutine is made, only the actual call statement is necessary. The original variable names are kept consistent to avoid having to make changes to the rest of the program. The compiler also produces information on the OpenCL work group topology that it expects each kernel to be run using. That is, it produces a value for the number of global work items that would be used when launching the code.

Calls to reduction subroutines require more than just the call statement. The principle behind parallel reductions is one of performing partial reductions and then reducing the results of the partial reduction. This is done by dividing the work among the available compute units on the device and then performing the same reduction on the results from the compute units. The final reduction must be performed by the host program and is done by a loop that iterates a number of times equal to the number of compute units and performs the reduction operation. Due to this device specific characteristic, the number of work items and the work group size produced as comments by the compiler are in terms of the number of compute units and the number of threads per compute unit for the target device.

Map Kernel Emission

For each of the OpenCLMap nodes in the transformed AST, a subroutine is produced with a name synthesised from the source line information of the original loop and the variables that are written to during the execution of the contained code. This synthesis is an attempt to produce unique names for every subroutine.

Each of the variables that are read and written during the execution of the map are included in the arguments to the subroutine and are declared using their declarations in the original source code. The subroutine also imports all of the modules imported by the original source file.

For the most efficient use of OpenCL and a target parallel device, it is necessary to remove all loops. It is therefore also necessary to be able to determine the values for the original loop iterator variables from the global id value of the current work item as otherwise the original operations that make up the map will not be correct. Expressions that yield the correct values for each of the loop iterator variables are therefore constructed and form part of the final subroutine code.

Finally, the original operations appear after the loop iterator expressions. The rest of the subroutine has been made in such a way to allow for the operations that formed the body of the original loop to appear verbatim in the final parallel map.

Reduction Kernel Emission

Like OpenCLMap nodes, every OpenCLReduce node in the AST becomes a subroutine of it's own. Reduction subroutines also share some characteristics with map subroutines, for example the presence of variable declarations, the make up of the arguments and the need for loop iterator expression construction. However, the final reduction subroutines are somewhat more complex than map subroutines to allow for effective use of hardware.

As mentioned earlier, the approach taken by our compiler for parallel reductions is one of dividing the problem into partial reductions and then reducing the results. There are two levels to this partial reduction scheme - each of the compute units performs a partial reduction on a portion of the whole dataset and each thread of each compute unit performs a partial reduction on a portion of the data points supplied to the parent compute unit. This method requires the introduction of new *local* variables that exist only within the scope of the subroutine and are only accessible from inside a compute unit. Additionally, since it is necessary for the host to perform the final reduction, new global arrays must also be used and added to the arguments to the subroutine. These global arrays will be assigned results from the partial reductions on each of the compute units.

The first step for the body of a reduction subroutine is to perform the partial reduction on a single thread. Using a loop that iterates over all of the data points assigned to that thread, a value for the partial reduction is calculated and assigned to a local array at a position determined by the thread's *local id*. The calculation is performed using the same code that appeared in the original source. The next step is to enact a *barrier* that forces all threads on a device to synchronise at that point in the program before allowing any of them to continue.

After synchronising, the second partial reduction runs. In this case, it is a loop that iterates over the partial results from each of the threads of that compute unit. The body of the loop is an assignment (or number of assignments) that performs only the primary reduction operation on the reduction variable. For example, if the original code involved multiplying a value by 3 and then adding it to the reduction variable, then the multiplication and addition operations would be performed by the thread level partial reduction. At the compute unit level, the operation would simply be the addition because it is the primary, associative operation performed on the reduction variable. The result from this second reduction is stored in a global array at a position calculated using the *group id* for this work item ready for the host to perform the final reduction later.

4. EVALUATION

To evaluate our work, we studied aspects of the compiler's output. Firstly we attempted to determine the success of the parallelism detection process with a variety of metrics that focused on the structure of the input and output source code. Our second evaluation involved analysing the performance benefit that generated OpenCL kernels produce over the original sequential programs.

4.1 Coverage

The focus of this work was to produce a parallelising compiler tailored to the structure and mechanisms employed by computationally intensive, meteorological programs such as large eddy simulators. To evaluate the success of this aim, we have attempted to show the *coverage* of our parallelism detection when the compiler is run against some subroutines of a Fortran version of the large eddy simulator[4]. In this context, coverage refers to the amount of a program that can be parallelised by our compiler. The results of this evaluation are shown in Table **??**.

The structure of this version of the LES involves a loop that iterates over time points and makes calls to a set of subroutines so that calculations for that point in time may be made. These subroutines make up the vast majority of the processing that the LES performs and are named *adam*, *bondv1*, *feedbf*, *les*, *les*, *press*, *vel2*, *velFG* and *velnw*.

The simplest of the metrics used here is the percentage line change. This value simply measures the percentage difference in number of lines when comparing the original Fortran source code and the output host code. For example, in the *press* subroutine, 56.1% of the original code has been removed or changed in some way by our compiler. The measure of *loops parallelised* refers to the percentage of loops from the original source that no longer exist in the output host code. Continuing with *press* as an example, 80% of the individual loops that were in the original subroutine do not appear in the output host code and instead their functionality is handled by generated parallel kernels. The measure of *loop clusters parallelised* is a similar metric but rather than considering each of the individual loops in the program, it considers clusters of nested loops (from the scope of the subroutine) as single entities. This measure is perhaps more insightful for this particular codebase as most of the computation is performed by nested loops that access multidimensional arrays. The final two fields in the Table show the number of map and reduce kernels that were produced from the original subroutines.

Almost all of the available parallelism in our test cases was correctly identified by our analysis techniques and converted to parallel OpenCL.

4.2 Performance

[4]LES repository: `https://github.com/wimvanderbauwhede/LES`

```
subroutine fortransum(input_array, tsize, result_out)

  integer :: tsize
  integer :: result_out
  integer, dimension(tsize) :: input_array
  result_out = 0
  do i = 1,tsize
    result_out = result_out + input_array(i)
  end do
end subroutine fortransum
```

Figure 1: Sequential Fortran subroutine that calculates the sum of all of the values of an array

Hardware

Our performance evaluation involved running both parallel and sequential versions of the a pair of subroutines and measuring the differences in run time. In order to fully demonstrate the performance benefits made available by parallelism, we ran the parallel versions of the subroutines on an nVidia GeForce GTX 590 graphics card. This device is quoted to have 1024 stream cores, a clock rate of 607 MHz and a memory bandwidth of 327.7 GB/s and represents an example of standard, commodity graphics hardware released in 2011[5]. While nVidia state that the device has 1024 cores, in OpenCL terms the device has 16 compute units and 32 processor elements per compute unit. During our evaluation, work group topologies are calculated using these compute unit and processor element values.

The sequential versions of the subroutines were run on a powerful Intel CPU, a Core i7-2600 with 4 cores, a quoted clock speed of 3.8 GHz (when only running a single core) and a memory bandwidth of 21 GB/s[6]. This CPU was also released in 2011 but has a high clock frequency that is still comparable to more recent Core i7 processors.

Evaluation

To evaluate the ultimate performance benefits that could be expected from the use of our compiler, we ran tests using two simple Fortran subroutines, one that would produce a map kernel and another that would produce a reduction kernel. Both subroutines were processed by our compiler to produce parallel Fortran using OpenCL. In order to test the new kernels, it was necessary to translate them to the C variant that OpenCL uses as kernel code and also to produce a C++ host program to drive the kernels. The C++ program also took advantage of the interoperability of C and Fortran to make calls to the original Fortran subroutines that the kernels were produced from. This allowed for the consistent use of the high resolution clock from the *chrono* library of C++ and made direct comparison of the parallel and sequential versions of the subroutines possible. Aside from time comparisons, it was also easy to check that the output from both versions of the subroutines was exactly the same.

The first program was written to calculate the sum of all of the elements of an array supplied as an argument and is shown in Figure **??**. This process is an example of a reduction, and was recognised as such when we ran the compiler against it to produce the code shown in Figure **??**. The set

[5]Full specifications of nVidia GeForce GTX 590: `http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-590/specifications`

[6]Full specifications of Intel Core i7-2600 `http://ark.intel.com/products/52213/Intel-Core-i7-2600-Processor-8M-Cache-up-to-3_80-GHz`

```
__kernel void reduce_result_out_8(__global int *input_array, int tsize,
      __global int *global_result_out_array) {
  int local_id;
  int group_id;
  int global_id;
  int local_id_fortran;
  int group_id_fortran;
  int local_chunk_size;
  int start_position;
  int local_result_out;
  int r_iter;
  int i_rel;
  int i;
  __local int local_result_out_array [64];

  local_id = get_local_id(0);
  group_id = get_group_id(0);
  global_id = get_global_id(0);


  local_id_fortran = (local_id + 1);
  group_id_fortran = (group_id + 1);
  local_chunk_size = ((tsize / 64) / 16);
  start_position = local_chunk_size * global_id;
  local_result_out = 0;
  for (r_iter=start_position;r_iter<=((start_position + local_chunk_size) -
       1);r_iter++) {
    i_rel = r_iter;
    i = (i_rel + 1);
    local_result_out = (local_result_out + input_array[i-1]);
  }
  local_result_out_array[local_id_fortran-1] = local_result_out;
  barrier( CLK_LOCAL_MEM_FENCE );
  local_result_out = 0;
  for (r_iter=1;r_iter<=64;r_iter++) {
    local_result_out = (local_result_out + local_result_out_array[r_iter
       -1]);
  }
  global_result_out_array[group_id_fortran-1] = local_result_out;
}
```

Figure 2: Generated parallel OpenCL kernel that calculates the sum of all of the values of an array

```
subroutine fortranmultiply(input_array, factor, tsize, result_array)

  integer :: tsize
  integer :: factor
  integer, dimension(tsize) :: input_array
  integer, dimension(tsize) :: result_array
  do i = 1,tsize
    result_array(i) = input_array(i)*factor
  end do
end subroutine fortranmultiply
```

Figure 3: Sequential Fortran subroutine that multiplies all of the values of an array by a factor and populates a resultant array

of tests involved varying the number of values that were to be summed by the kernel and subroutine. Each test case was run 20 times and the mean value taken to form the results of this test. Figure **??** shows the speed up in performance that our generated kernel displayed for this example reduction over all of our test cases. Values from all tests of this kernel can be seen in Tables **??**, **??** and **??**.

To test the performance of the map pattern recognised by our compiler, we wrote another simple Fortran subroutine (Figure **??**). In this case, the subroutine was designed to multiply every value in an input array by a certain factor and return the result in a new array. As before, the subroutine was processed by our compiler which correctly identified the subroutine as a map and output the code shown in Figure **??**. The same method, of varying input array sizes, was used for this test as with the previous and a graph of achieved speed ups over all test cases is shown in Figure **??**. Results from this kernel's performance tests are shown in Tables **??**, **??** and **??**.

4.3 Discussion

Our coverage tests present some values that point to how much of the original codebase of the LES can be parallelised by our compiler. The first measure, the line change value, could be considered rather arbitrary as the number of lines in a program does not necessarily relate to the amount of processor time or computational power that is needed to

Subroutine	*Line Change*	*Loops Parallelised*	*Loop Clusters Parallelised*	*Parallel Maps*	*Parallel Reductions*
adam	60.3%	100.0%	100.0%	1	0
bondv1	69.4%	82.8%	84.6%	6	1
feedbf	57.6%	100.0%	100.0%	1	0
les	81.3%	100.0%	100.0%	3	0
press	56.1%	80.0%	88.9%	3	2
vel2	71.5%	100.0%	100.0%	10	0
velFG	51.6%	100.0%	100.0%	1	0
velnw	60.8%	100.0%	100.0%	1	0

Table 1: Table showing detection of parallelism across LES codebase

Domain Size	*1,024*	*2,048*	*4,096*	*8,192*	*10,240*	*12,288*	*14,336*	*16,384*	*18,432*	*20,480*
Sequential (μs)	2.23	4.33	8.63	17.09	21.35	25.62	30.01	34.65	38.73	42.76
Parallel (μs)	38.91	39.15	39.61	39.80	41.80	40.50	41.44	41.28	42.46	43.47
Speed up	0.06	0.11	0.22	0.43	0.51	0.63	0.72	0.84	0.91	0.98

Table 2: Performance results from original sum subroutine (Figure ??) and generated parallel sum kernel (Figure ??) when run with small data sets

Domain Size ($\times 10$)	*1,024*	*2,048*	*4,096*	*8,192*	*10,240*	*12,288*	*14,336*	*16,384*	*18,432*	*20,480*
Sequential (μs)	21.58	42.86	86.04	171.49	215.49	259.12	297.01	339.32	380.09	420.27
Parallel (μs)	40.51	42.43	49.07	61.65	68.14	71.12	79.20	85.00	92.01	100.03
Speed up	0.53	1.01	1.75	2.78	3.16	3.64	3.75	3.99	4.13	4.20

Table 3: Performance results from original sum subroutine (Figure ??) and generated parallel sum kernel (Figure ??) when run with medium sized data sets

Domain Size ($\times 10^3$)	*1,024*	*2,048*	*4,096*	*8,192*	*10,240*	*12,288*	*14,336*	*16,384*	*18,432*	*20,480*
Sequential (ms)	2.12	4.34	9.21	17.49	21.96	26.19	30.53	34.93	39.45	43.68
Parallel (ms)	0.39	0.64	1.20	2.35	2.94	3.36	4.18	4.77	5.41	5.49
Speed up	5.46	6.82	7.69	7.44	7.48	7.80	7.31	7.33	7.29	**7.95**

Table 4: Performance results from original sum subroutine (Figure ??) and generated parallel sum kernel (Figure ??) when run with large data sets

Domain Size	*1,024*	*2,048*	*4,096*	*8,192*	*10,240*	*12,288*	*14,336*	*16,384*	*18,432*	*20,480*
Sequential (μs)	2.50	5.98	12.23	26.56	32.29	39.13	44.84	51.25	58.57	65.20
Parallel (μs)	41.67	43.60	46.57	55.55	57.88	61.56	65.58	68.49	73.78	77.40
Speed up	0.06	0.14	0.26	0.48	0.56	0.64	0.68	0.75	0.79	0.84

Table 5: Performance results from original multiply subroutine (Figure ??) and generated parallel multiply kernel (Figure ??) when run with small data sets

Domain Size ($\times 10$)	*1,024*	*2,048*	*4,096*	*8,192*	*10,240*	*12,288*	*14,336*	*16,384*	*18,432*	*20,480*
Sequential (μs)	32.28	65.41	130.38	259.81	325.51	386.58	447.85	509.93	570.94	636.33
Parallel (μs)	58.70	76.51	114.02	187.64	225.85	261.95	298.48	334.09	372.53	409.28
Speed up	0.55	0.85	1.14	1.38	1.44	1.48	1.50	1.53	1.53	1.55

Table 6: Performance results from original multiply subroutine (Figure ??) and generated parallel multiply kernel (Figure ??) when run with medium sized data sets

Domain Size ($\times 10^3$)	*1,024*	*2,048*	*4,096*	*8,192*	*10,240*	*12,288*	*14,336*	*16,384*	*18,432*	*20,480*
Sequential (ms)	3.25	6.55	13.15	26.46	32.82	39.06	45.51	51.99	58.25	64.57
Parallel (ms)	1.34	2.64	5.23	10.42	13.03	15.61	18.21	20.80	23.43	25.99
Speed up	2.43	2.48	2.51	**2.54**	2.52	2.50	2.50	2.50	2.49	2.48

Table 7: Performance results from original multiply subroutine (Figure ??) and generated parallel multiply kernel (Figure ??) when run with large data sets

```
__kernel void map_result_array_8(__global int *input_array, int factor, int
       tsize, __global int *result_array) {
  int global_id;
  int i_rel;
  int i;

  global_id = get_global_id(0);

  i_rel = global_id;
  i = (i_rel + 1);

  result_array[i-1] = input_array[i-1] * factor;
}
```

Figure 4: Generated parallel OpenCL kernel that multiplies all of the values of an array by a factor and populates a resultant array

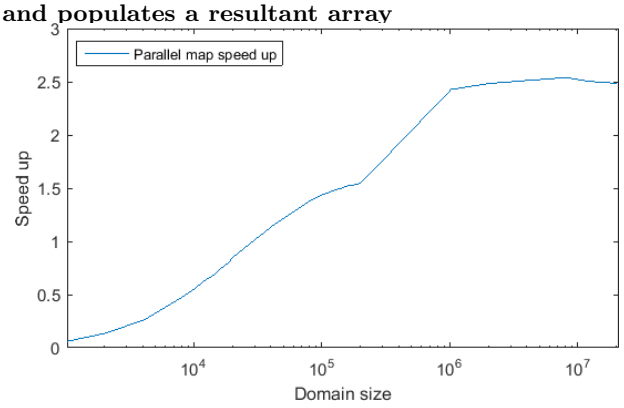

Figure 5: Graph showing the speed up achieved by the parallel version of a map kernel

complete the program. These values were included simply to help illustrate the changes that our compiler makes.

The *loops parallelised* and *loop clusters parallelised* metrics are more informative of the changes made to the actual structure of the original program. Since all of the heavy computation will take place in loops, it is clear that parallelising all of the loops means that all of the intensive elements of the program have been parallelised.

The loop clusters metric was introduced as a measure specific to the LES. All of the major elements of the program are performed in nested loop clusters and so it seemed intuitive to include a measure focused on loop clusters.

The coverage tests we have performed show that our compiler parallelises the vast majority of computation that takes place as part of the LES. There were two test cases that did not have all of their loops parallelised by our compiler, *bondv1* and *press*. For *bondv1*, the missing 16.4% is attributed to five loops nested into two loop clusters. For the first loop cluster, a triple nested cluster, the compiler recognises that the innermost loop is parallelisable as a map under our conditions but that the outer two loops are not. The outer loops cannot be mapped because they have loop carried dependencies and access non temporary variables at locations not specified by the loop iterator variables. The second loop cluster cannot be parallelised for the same reasons. In *press*, a single loop cluster cannot be parallelised because of a loop carried dependency.

While our compiler does not recognise some loop clusters as being completely parallel, it is not true to say that they are not parallel, just that our conditions are too strict. All three of the loop clusters that are not made parallel by our compiler in the coverage tests can be seen to be parallel if the range of values for the loop iterators are considered. Our compiler currently does not support this form of analysis as it lacks the functionality to evaluate expressions.

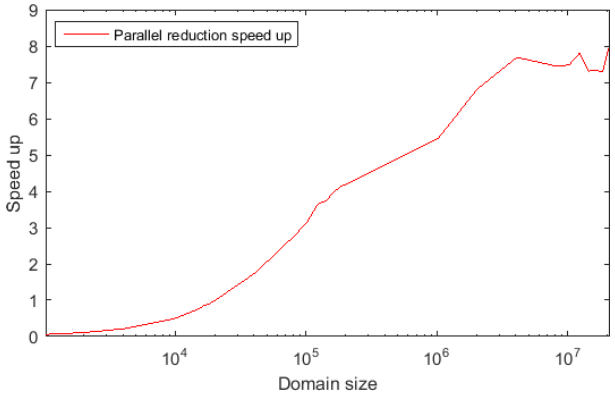

Figure 6: Graph showing the speed up achieved by the parallel version of a reduction kernel

The results of the performance tests that formed part of this evaluation are indicative of the benefit that our compiler produces. Fortran subroutines that were parallelised into OpenCL kernels show drastic speed ups when it is considered that these kernels perform few computations and are therefore memory bound. With this bound in mind, it is unlikely that hand written parallel kernels would perform any better than our automatically generated kernels. The fact that these speed ups from parallelism can be so easily produced demonstrates the gain in user productivity and code performance that can be achieved through use of our compiler.

5. CONCLUSION

We have developed a source to source compiler that automatically parallelises meteorological Fortran programs using OpenCL. Our compiler automatically recognises opportunities for parallelism by detecting patterns in the structure of the input source code, without the need for pragmas or compiler directives and applies parallel algorithmic skeletons to that program's abstract syntax tree. The transformed abstract syntax tree, along with the original source code, is used to produce a parallel, OpenCL version of the original program.

Our evaluation shows that our compiler successfully extracts almost all of the available parallelism from a typical atmospheric simulator, a *large eddy simulator*, written in Fortran. We have also shown the performance benefits that are available when using our compiler to automatically generate OpenCL code are substantial. When running reductions on parallel GPU hardware, our simple tests showed up to 8x speed ups when compared to sequential versions of the same program.

The compiler, while still a proof of concept, allows a climate scientist the ability to leverage powerful, affordable GPU hardware when performing computationally intensive simulations. A user need not have any knowledge of parallel programming or indeed the underlying hardware that they intend to use. The ultimate benefit is increased productivity through the removal of the often challenging task of writing effective programs for highly parallel hardware and thus an increase in the output of high quality research.

Future Work. As future work we intend to extend our compiler to perform more complex analysis, including the implementation of an expression evaluator that would al-

low for consideration of the exact variables and locations that are accessed during a loop iteration and thus improve dependency analysis. Also, the compiler could be extended to recognise and parallelise patterns such as the *pipeline* pattern presented by Marques et al. [**marques2013algorithmic**].

Outside of extending the parallelism detection ability of the compiler, code emitters for other parallel programming tools such as CUDA or OpenMP could be added without any major change to the core functionality of parallelism detection by the compiler. Similarly, emitting code that is more suited to other types of parallel architecture such as hardware accelerators or FPGAs is a desirable feature.

This work highlights the need for an OpenCL API for Fortran as there exists a large codebase in climate science alone that would benefit greatly. Therefore, the implementation of such an API is also future work.