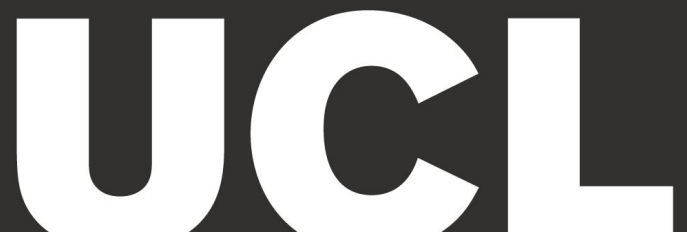
UCL DEPARTMENT OF
COMPUTER SCIENCE



Research Note

RN/18/04

Towards automatic generation and insertion of OpenACC directives

April 12, 2018

Bobby R. Bruce Justyna Petke

Abstract

While the utilisation of hardware accelerators, like GPUs, can significantly improve software performance, developers often lack the expertise or time to properly translate source code to do so. In this paper we highlight two approaches to automatically offload computationally intensive tasks to a system's GPU by generating and inserting OpenACC directives; one using grammar-based genetic programming, and another using a bespoke four stage process. We find that the grammar-based genetic programming approach reduces execution time by 2.60% on average, across the applications studied, while the bespoke four-stage approach reduces execution time by 2.44%. Despite this, our investigation shows a handwritten OpenACC implementation is capable of reducing execution time by 65.68%. Comparing the differences, we identified a promising avenue for future research: combining genetic improvement with better handling of data to and from the GPU.

1 Introduction

As the power of a single processing core reaches its limit, modern computer systems have become increasingly reliant on multicore architectures and accelerators; devices such as GPUs, FPGAs, and cryptography co-processors that are capable of improving the performance of specific computational tasks.

CUDA [13] and OpenCL [19] are languages that have been introduced to allow the development of software for GPUs. The VHSIC Hardware Description Language exists for FPGAs [11], and various frameworks and libraries exist to utilise multicore CPUs [7, 8]. However, utilising these languages, libraries, and frameworks is difficult for most developers as they do not lend themselves to traditional programming paradigms. In the domain of optimisation, software written to be executed sequentially (as is in languages such as C/C++) cannot easily be translated to run on these targets like multicore CPUs or GPUs as these architectures can only increase performance via parallelisation. Indeed, translation may be impossible — some code is simply dependent on one instruction being executed after another. Learning to use these languages, frameworks, or libraries correctly is a large investment. In many cases, utilisation can be of considerable benefit to software performance but it is not economical due to the costs of human developers and the training they require.

In response to this, directive-based approaches have been developed. These allow engineers to create software in a manner in which they are comfortable and later add directives that inform the compiler to offload certain segments of code to other processing cores and/or hardware accelerators. Though simpler to utilise than CUDA, OpenCL, or other languages and frameworks, implementing these directives still requires training and incurs a cost in regards to developer time as the process of adding these directives to software is one of trial and error. Our goal in this research is to develop techniques to automatically insert these directives without any human intervention.

In this investigation we focused on optimising code to utilise the system's GPU. As such we studied inserting OpenACC [21] directives as OpenACC is, at the time of writing, the state-of-the-art when it comes to directive based GPGPU programming. It has been shown superior to its main rival, OpenMP [7] in this domain[1] [18].

To the best of our knowledge, there is only one other approach that automatically inserts OpenACC directives — a source-to-source compiler module called DawnCC [12]. DawnCC uses established static analysis techniques to determine where to insert OpenACC directives safely while incorporating range analysis to determine the range of arrays to copy from and to the system's GPU. However, we found that when DawnCC was run on the NAS-NPB benchmark suite (the suite of applications we later use to evaluate our approaches), DawnCC increased execution time in six of the seven applications (and could not influence execution time by a statistically significant extent in the remaining one)[2].

[1] It should be noted, both OpenMP and OpenACC are similar standards so the techniques described in this paper could be adapted to insert OpenMP directives if desired.

[2] This short investigation is outlined, in greater detail, in Appendix B.

The reason for this is simply offloading code to the GPU does not guarantee faster code. For each parallelised region of code, there is an overhead where information must be transferred to the GPU for processing then transferred back upon completion. It is not at all uncommon for this transfer overhead to nullify any improvements that may be gained form parallelisation. This overhead is difficult, if not impossible, to determine at compile time.

So, while an important contribution, the techniques outlined in this paper differ to DawnCC in that they utilise *genetic improvement* [17]; a sub-domain of search-based software engineering [9] which applies search-based techniques to modify and improve software with respect to some user-defined objective. In genetic improvement the problem faced by DawnCC is not so much a problem as the target software is run and measured during optimisation using tests representative of usage during optimisation. Additionally, compilers must preserve the semantics of source code regardless of whether the semantics are efficient or necessary. Genetic improvement, on the other hand, only preserves the semantics specified by the user through a fitness objective (typically in an implicit manner via tests). This allows for more avenues of optimisation. We think the application of genetic improvement to the automatic generation and insertion of OpenACC directives is a worthwhile endeavour and we are therefore the first to apply genetic improvement to parallelise code by the automatic generation and insertion of directives.

We investigated two genetic improvement approaches to automatically generate and insert OpenACC directives. One utilises grammar-based genetic programming [20] (GB-GP) to produce directives which are then inserted into the target source code with the execution time of the modified software (once compiled and run on a test input) used as the GB-GP's fitness measure. We refer to this as *GB-GP-Parallelisation*. In the other takes a more bespoke approach. It inserts OpenACC directives to source code and tunes them in four separate steps, each of which uses either a greedy algorithm or an evolutionary strategy [3]. We refer to this as *Four-Stage-Parallelisation*.

We evaluated our approaches on a sequential variant of the NAS Parallel Benchmark suite [4] provided by the Center for Manycore Programming at Seoul National University [6]. We refer to this collection of applications as the SNU-NPB suite. We ran both the GB-GP-Parallelisation and Four-Stage-Parallelisation approaches on each application within the SNU-NPB suite to produce variants which contain OpenACC directives. We compared the execution time of the sequential applications against these variants when running a test input to evaluate our approaches.

In order to make a truly fair evaluation as to the effectiveness of our approaches we needed a baseline truth; knowledge of what is achievable when attempting to optimise these sequential applications by inserting OpenACC directives. Fortunately found such a baseline. Pino et al. parallelised the SNU-NPB suite using OpenACC directives as part of an investigation comparing OpenACC against OpenMP [18]. They have since made this hand-implemented OpenACC variant public [2]. We refer to this as the SNU-NPB-ACC suite. We use the performance of the SNU-NPB-ACC suite as a 'best-case scenario' proxy

for our automatic OpenACC insertion techniques. With this setup, we ask the following research questions:

RQ1 How effective is GB-GP-Parallelisation?

- **RQ1a** What execution time reductions are achievable when using GB-GP-Parallelisation?
- **RQ1b** How do the reductions in execution time compare to handwritten OpenACC implementations?

RQ2 How effective is Four-Stage-Parallelisation?

- **RQ2a** What execution time reductions are achievable when using Four-Stage-Parallelisation?
- **RQ2b** How do the reductions in execution time compare to handwritten OpenACC implementations?

RQ3 What differs between the solutions produced by GB-GP-Parallelisation, Four-Stage-Parallelisation, and the handwritten OpenACC implementations?

We found that neither techniques were capable of producing substantial savings in execution time. GB-GP-Parallelisation reduced execution time by 2.79% on average, while Four-Stage-Parallelisation reduced execution by 2.44%. In contrast, the handwritten SNU-NPB-ACC implementations reduces execution time by 65.68% on average. We found no cases where the solutions produced by GB-GP-Parallelisation and Four-Stage-Parallelisation could improve upon both the sequential and hand-implemented variants of an application.

When comparing the solution we found that the handwritten implementations in the SNU-NPB-ACC suite utilised the OpenACC `#pragma acc data` directive to a significantly greater extent than in either of our approaches. These directives manage how and when data is moved to and from the GPU; a costly exercise. When we removed these directives from the handwritten implementation, the execution time of the applications increased significantly (from just over 1.9 seconds to 3463.23, just under an hour, in one case), showing that these directives are extremely important when maximising the reducing execution time via parallelisation. We therefore advise that future researchers put more emphasis on optimising the flow of data to and from the hardware accelerators.

2 Setup

In this section we discuss the design of both the GB-GP-Parallelisation and Four-Stage-Parallelisation techniques, the environment in which they were run, the applications in which they targeted, and the methodology we adhered by to obtain the data necessary to answer the research questions.

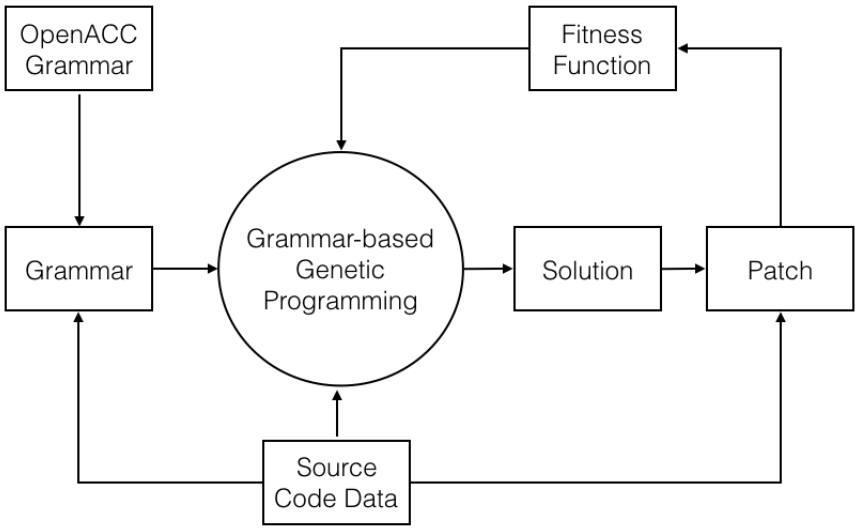


Figure 1: The Architecture of GB-GP-Parallelisation

```
<start>:: <start> | <start> | <directive>
<directive> ::= "#pragma acc parallel loop " <loop_line_number>
         | "#pragma acc wait " <line_number>
<loop_line_number> ::= "34@FileB.c" | "55@FileA.c"
<line_number> ::= "11@FileA.c" | "40@FileB.c" | "55@FileC.c"
```

Figure 2: A heavily-abridged example of the OpenACC grammar in Backus Normal Form (BNF)

2.1 GB-GP-Parallelisation

Figure 1 shows the basic architecture of GB-GP-Parallelisation. At its core is a grammar-based genetic programming algorithm which produces a solution that is then translated into a patch that may be applied to the target source code to insert OpenACC directives. This patch is evaluated by a fitness function, and this fitness is returned to the grammar-based genetic programming algorithm. A grammar-based genetic programming algorithm functions like a traditional genetic programming (GP) algorithm [5] but has fixed rules (i.e. a grammar) which describes how terminals and non-terminals may join in the tree. In a traditional grammar-based GP setup they are two inputs: the grammar, and a fitness function to evaluate the solutions it produces. Our setup is more convoluted as it requires specific data from the target source code to create both the grammar and to generate a patch from a solution output by the grammar-based GP.

Figure 2 shows an abridged snippet of the OpenACC grammar in Backus Normal Form (BNF), like that we use in the GB-GP-Parallelisation setup. Grammar-based GP searches the set of solutions achievable within its grammar and, by design, is incapable of producing grammatically invalid solutions. As an example, for the grammar in Figure 2, `#pragma acc parallel loop 11@FileA` can never be generated as `11@FileA.c` is a production of `<line_number>`, not `<loop_line_number>`.

In our work the grammar is considerably more complex than this. The data necessary to produce a solution that will then be converted into a patch requires two components: what we call the 'OpenACC grammar' (which is a constant across all target source code, shown in Appendix A), and source code data. The OpenACC grammar is incomplete as, while it contains rules on how OpenACC directives must be structured for the compiler to interpret, it is missing production rules on where to insert these directives. GB-GP-Parallelisation appends these production rules to the OpenACC grammar during execution based on data extracted from the target source code. The full grammar (the OpenACC grammar plus the source code data) is thereby unique for each application. These appended production rules are the source code line numbers (`<line_number>`), the location of `FOR` loops (`<loop_line_number>`), the location of the first statement within a `FOR` loop (`<top_loop_line_number>`), and the location of functions (`<function_line_number>`). In our setup the grammar-based GP will output solutions such as `#pragma acc parallel loop 12@FileA.c`. The postfix, `12@FileA.c`, is information unique to that application; a valid insertion point for the directive in this case. These solutions are then parsed into patches. The `#pragma acc parallel loop 12@FileA.c` example would result in a patch that inserts `#pragma acc parallel loop` at line `12` in `FileA.c`.

With the aforementioned production rules appended, the grammar-based GP can produce solutions, though translation is required to transform them into software patches. Some OpenACC directives declare how program variables are transferred to and from the GPU and main system memory. The patches therefore require the names of variables. In the grammar, variables are represented as integer placeholders, ranging from 1 to 100. These placeholders are translated into variable names when the solution is transformed into a patch. To translate an integer placeholder i into a variable we obtain a sorted vector V of all variables within the directive to be inserted's scope. We then select V_j from this vector where $j = i \mod |V|$. For example, the partial solution `#pragma acc parallel loop copyin(34)` with the vector of variables V equal to `[varA, varB, varC, varD]`, would select the $34 \mod |V|$th element in V (where 0 is the first element). In this case $34 \mod |V| = 2$, and thus `varC` would be chosen, resulting in the solution being translated to `#pragma acc parallel loop copyin(varC)`.

Finally there are circumstances when our grammar may introduce new program scopes (i.e., curly brackets in C/C++). We represent the location of the end of these scopes as a placeholder in the OpenACC grammar. Again, these range from 1 to 100. A solution produced by the grammar may be

`#pragma acc kernels \n{35 15@FileB.c` which would be translated into a patch by inserting `#pragma acc kernels \n{` at line 15 in `FileB.c`, then inserting the closing bracket, `}`, 35 statements after this insertion, skipping inner scopes (such as `FOR`, `WHILE`, and `IF` statements). If the end of the current scope is reached, we start over from the directive insertion point (in this example case, line 35).

To implement our grammar-based genetic programming approach, we used Epochx 1.4.1 [15], an open source genetic programming framework, written in Java, which supports grammar-based genetic programming. As genetic programming algorithms can be setup in any number of ways, it is important to state how that contained in Epochx 1.4.1 functions.

Algorithm 1 The Genetic Programming Algorithm

Require: G, the number of generations, $G \in \mathbb{N}$
C, the crossover rate, $C \in \mathbb{R} \wedge 0 \leq C \leq 1$
M, the mutation rate, $M \in \mathbb{R} \wedge 0 \leq M \leq 1$
S, the population size, $S \in \mathbb{N}$
T, the tournament size, $T \in \mathbb{N}$
initialise_population(S), returns an initial population of size S
evaluate(P), evaluates all the solutions in the population P
select(P,T), Tournament selection of size T on population P
crossover(p_1, p_2), crossover using parents p_1 and p_2
mutate(p), returns a mutant of solution p
get_random$()$, returns a random number $r : r \in \mathbb{R} \wedge 0 \leq r \leq 1$

1: $P \leftarrow$*initialise_population*(S)
2: *evaluate*(P)
3: **for** $1 \ldots G$ **do**
4: $P' \leftarrow \{\}$
5: **while** $|P'| < |P|$ **do**
6: $r \leftarrow$ *get_random*$()$
7: **if** $r < C$ **then**
8: $p_1 \leftarrow$ *select*(P,T)
9: $p_2 \leftarrow$ *select*(P,T)
10: $P' \leftarrow \{P'\} \cup \{$*crossover*$(p_1, p_2)\}$
11: **else if** $r < (C + M)$ **then**
12: $p \leftarrow$*select*(P)
13: $P' \leftarrow \{P'\} \cup \{$*mutate*$(p)\}$
14: **else**
15: $P' \leftarrow \{P'\} \cup \{$*select*$(P,T)\}$
16: **end if**
17: **end while**
18: $P \leftarrow P'$
19: *evaluate*(P)
20: **end for**
21: **return** x, where $x \in P \wedge x.fitness \neq -1 \wedge \forall p \in P : x.fitness \leq p.fitness$

Algorithm 2 Generating the Initial Population

Require: S, the population size, $S \in \mathbb{N}$
$mutate(p)$, returns a mutant of solution p
$get\_initial\_solution()$, returns a single initial solution
$get\_random\_solution()$, returns a random solution

1: $P \leftarrow \{\}$
2: $i \leftarrow 0$
3: $N \leftarrow 3S$
4: **while** $|P| < S \wedge i \leq N$ **do**
5: $P \leftarrow \{P\} \cup \{get\_initial\_solution()\}$
6: $i \leftarrow i + 1$
7: **end while**
8: $i \leftarrow 0$
9: **while** $|P| < S \wedge i \leq N$ **do**
10: $p \leftarrow get\_initial\_solution()$
11: $P \leftarrow \{P\} \cup \{mutate(p)\}$
12: $i \leftarrow i + 1$
13: **end while**
14: **while** $|P| < S$ **do**
15: $P \leftarrow \{P\} \cup \{get\_random\_solution()\}$
16: **end while**
17: **return** P

Algorithm 1 outlines the genetic programming algorithm used in GB-GP-Parallelisation. Conforming to the grammar, a population P is initialised via the function *initialise_population*. The population is then evaluationed via the *evaluate* method, giving each solution in the population a fitness value. The algorithm then iterates through a number of generations G. At the start of each new generation another population P' is derived from the previous generation P using crossover, mutation, and selection; the proportion of each based on the crossover rate C, and mutation rate M. Both *crossover* and *mutate* require solutions to be selected via *select*, a tournament selection of size T which selects based on the fitness of the solutions in the population P. Once the generation of P' is complete, P is replaced by P' and the population is evaluated. The best solution from the final generation, based on the solutions' fitness, is then returned as the final, 'champion', solution.

The functionality of *initialise_population* is shown in Algorithm 2. The *initial_population* algorithm starts by attempting to populate set P with those generated by *get_initial_solution* which iterates through the target source code's `FOR` loops returning a solution which adds the `#pragma acc parallel loop` directive to this location. As it is possible that the number of `FOR` loops is less than the population size, and duplicates are not permitted, we limit the number of calls to *get_initial_solution* to three times the population size (N). If, after this, the population size is not met, *get_initial_solution* is called with its output mutated via the mutation operator. Again, this attempt is limited

```
<start>::= <start> <start> | <directive>
<directive>::= "#pragma acc " <choice>
<choice>::= "parallel "  <parallel> ...
| "loop " ... <loop_line_number>
| "parallel loop " ... <loop_line_number>
| "kernels " ... <line_number>
| "kernels loop " ... <loop_line_number>
| "data " ... <line_number>
...
```

Figure 3: An abridged version of the first three production rules in the OpenACC grammar

by three times the population size iterations (though given the search space of mutating one directive being high, this is rarely met in practise). Finally, if the population size is still not met, the rest of the initial population is filled with truly random solutions (via *get_random_solution*).

Our crossover operator takes two solutions as parents (selected using tournament selection) then produces a child with each directive, within each parent, having a 50% change of being included in the child solution. Figure 3 shows how our OpenACC grammar permits this type of crossover as directives (`<directive>`) may be taken away or added indefinitely.

Our mutation operator is based on the mutation operator first outlined by Whigham when he proposed grammar-based genetic programming [20]. Whigham mutation randomly selects a subtree and replaces it with a randomly generated (and grammatically correct) alternative. In our customisation, we do this 50% of the time. In the other 50% of the time we add a new directive to the solution, produced by calling *get_initial_solution*. This is our way of introducing (or reintroducing) sensible genetic material into the population.

When evaluating a solution produced by the grammar-based GP, we translate it into an UNIX patch. However, before doing this we run some checks on the solution. There are some instances where the grammar produces incorrect or inefficient solutions that are easy to fix. Our goal with this fix step is to reduce the number of ineffective evaluations. First we check that no two directives share the same insertion point. If this is found we select one and uniformly select another location (conforming to the grammar) within the source code. We iterate through all the directives within the solution until we achieve a pass in which all directives have a unique insertion point. As it is feasible that all insert locations are occupied, we only allow a maximum of five passes over the same solution. If this maximum is met, the fixing is declared to have failed which automatically results in the solution being rejected, with a fitness value equal to that if it did not meet our fitness function hard constraints. Next we check to ensure that any instance of `#pragma acc loop` is contained within a `#pragma acc kernels loop`, `#pragma acc kernels`, or `#pragma acc parallel loop`

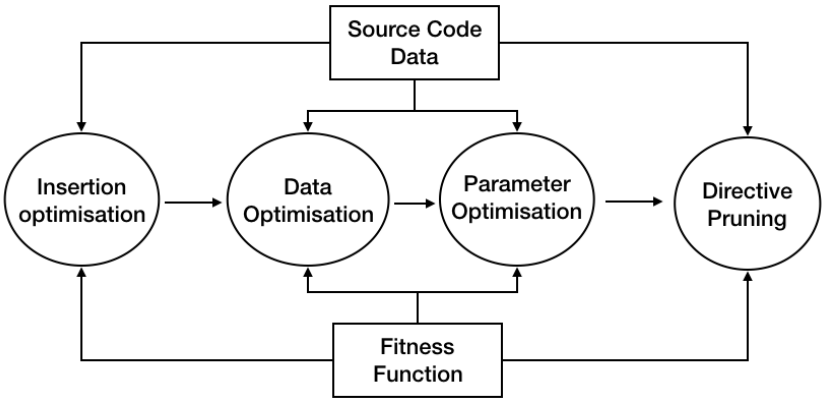


Figure 4: The Architecture of Four-Stage-Parallelisation

construct. `#pragma acc loop` is inert when not present within these and, if such an instance is found, we convert it to `#pragma acc parallel loop` so it has an opportunity to optimise. Likewise, a `#pragma acc parallel loop` directive cannot be nested within a `#pragma acc parallel loop` or `#pragma acc kernels` construct. In such an instance the directive is changed to `#pragma acc loop`.

Once this 'fixing' step is complete the framework translates the solution into a UNIX patch. This is then passed to a script, bespoke to each application, which returns the fitness value. The contract of this script is it must return a positive double value, and this double tends towards zero for fitter solutions. A value of '-1' is returned if the solution breaks a hard constraint. In all the applications we target, we write the fitness function to return the execution time of the applications. If application cannot be compiled, produces an invalid output, or reaches a timeout (10 minutes), we return '-1'. In our setup this is translated to a fitness value equal to `java.lang.Double.MAX_VALUE` within the Epochx GB-GP evaluation procedure. When the script returns the fitness value for that patch, we give the corresponding solution that fitness and let the GP continue.

2.2 Four-Stage-Parallelisation

Figure 4 shows the basic architecture of Four-Stage-Parallelisation. The optimisation happens over four distinct stages which we shall explain in this Section. Optimisation starts with *insertion optimisation* then moves through to *directive pruning*, *parameter optimisation*, and, finally, *delta debugging*. The origin of this design is that, unlike the process of GB-GP-Parallelisation, it more accurately represents the steps undertaken by a human implementing OpenACC directives. Adding a `#pragma acc parallel loop` directive to parallelise a `FOR` loop can increase execution time by a large amount until the correct parameters

are added. Humans are aware of this and, as such, common advice on implementing OpenACC is to do so in stages, being aware that the product at the end of the earlier stages may be worse than the original, unmodified software. The different stages in the Four-Stage-Parallelisation take this consideration into account.

The first stage, *insertion optimisation*, is a greedy approach to annotating `FOR` loops as parallelisable (via the insertion of `#pragma acc parallel loop` directives). Our goal in this stage is not to optimise for execution time but instead maximise the number of `FOR` loops that are parallelised. For each `FOR` loop within the program, a `#pragma acc parallel loop` directive is added, then the fitness function is run. In our setup a fitness function of '-1' indicates a failure to compile, a failure to preserve semantics, or a timeout. The *insertion optimisation* keeps any `#pragma acc parallel loop` insertions *if* the fitness function returns any value other than '-1'.

It should be noted that in the case of nested loops, an outer loops is always tested before its inner loops. As inserting a `#pragma acc parallel loop` directive cannot appear within a `FOR` loop already parallelised by a `#pragma acc parallel loop` directive (the compilation will fail), the framework has a bias to the parallelisation of outer loops. This is favourable as it results in more of the program being parallelised. When this stage is complete the solution with the most parallelised `FOR` loops (while preserving semantics and avoiding timeouts) is passed to the second stage.

The second stage, *data optimisation*, optimises the handling of variables that are initialised before a parallelised `FOR` loop structure but utilised within. In OpenACC each of these variables may be declared as either `copy`, `copyin`, `copyout`, `create`, or `present`. `copy` for when the variable must be copied into the GPU at the start of the parallelised `FOR` loop and copied out at the end. `copyin` for when the variable only needs to be copied into the GPU before computation but not copied back to main memory after. `copyout` for when the variable only needs to be copied back to main memory after (the variable is created on the GPU, but no value is moved from the system's main memory, it is assumed the value will be set within the GPU). `create` for when the variable should be created within, and neither copied to or from, the GPU. Finally there is `present` which informs the compiler that the variable is already within the GPU and no action is needed. Without declaring a variable, the decision is left to the compiler. These declarations are appended to the end of OpenACC directives, for example: `#pragma acc parallel loop copy(variable)`.

To optimise this we represent each variable decision as a gene within a genotype. The value (or allele) of each gene is the corresponding variable's status within a specific parallelised `FOR` loop (we include a 'no status set' option to leave the decision to the compiler). We carry out a (1+1) evolutionary strategy (ES) [3] on this representation, with a mutation rate (the probability that any gene in the genotype is mutated while generating a variant) of $1/l$ where l is the number of genes/variables. A mutation operation on any gene will change its value to another state uniformly selected from all available states. We start with an initial genotype with all its genes having a 'no status set' value. In this

stage the fitness function is utilised in full (same fitness function as used for the GB-GP-Parallelisation). That is, we attempt to reduce the execution time. We take the best solution found during this and pass it forward to the next stage.

The *parameter optimisation* stage functions in a similar manner to *data optimisation* stage. For each `#pragma acc parallel loop` directive there are three parameters which we optimise — `num_gangs`, `num_workers`, and `vector_length`. In the OpenACC model of parallelisation there are three levels: 'gang', 'worker', and 'vector' The vector layer functions as a SIMD parallelisation; individual instructions working over multiple data elements. The `vector_length` is the number of data elements that may be operated on with the same instruction. The worker layer 'works' an individual vector. The `num_workers` specifies how many workers there are within a gang. A gang is a collection of workers that share a common cache memory. `num_gangs` specifies the number of gangs. Their is no synchronisation or data communicated between gangs, each works completely independently. This model of parallelisation has been designed so OpenACC may target many different types of hardware — different types of GPUs as well as other hardware accelerators. Therefore, for each `FOR` loop parallelisation, for each hardware target, there is an optimal setting for the `num_gangs`, `num_workers`, and `vector_length` parameters. If these parameters are not specified they are determined by the compiler though they are seldom optimal.

In our experiments we set each of these parameters to 2^X where $X \in \mathbb{N} \wedge X \leq 10$. Like before, we have a special 'value not set' status to indicate the setting of a parameter is to be left to the compiler. As in the *data optimisation* stage, we represent the space of parameter settings as a series of genes. Every parallelised `FOR` loop has three genes within the genotype, one gene for each of the three parameters (`num_gangs`, `num_workers`, and `vector_length`). We then use a (1+1)-ES with a mutation rate of $1/l$ where l is number of genes/parameters to tune. The fitness function is the same as in the *data optimisation* stage; optimising the execution time while preserving program semantics. All the genes in the initial genotype are set to 'value not set' (i.e. the decision is left to the compiler). As in the previous stages, we take the best solution found and pass it forward to the next, and final, stage.

The goal of the final stage, *directive pruning*, is to remove any directives which are not decreasing the programs execution time, even after optimising the handling of data and their parameters. We do this in a greedy manner. First we measure the fitness of the current, full, solution, f_c. For each parallelised `FOR` loop, we remove the `#pragma acc parallel loop...` directive and measure the fitness, f, of the solution. If $f \leq f_c$ then $f_c = f$. Otherwise, the directive is returned to the solution.

It should be noted that *directive pruning* may return a solution where no loops are parallelised, or even a solution in which the execution time is greater than the original. The directives are not completely independent. We find there is a synergistic effect where a set of directives must all be removed to reduce execution time while removing a subset of these directives increases execution time. This stage outputs the final solution.

When running both GB-GP-Parallelisation and Four-Stage-Parallelisation

we must allocate a certain number of evaluations. While in GB-GP-Parallelisation the number of evaluations can be set by altering the population size and number of generations, in Four-Stage-Parallelisation, the division of evaluations between stages must be carefully considered. To do so we first acknowledge that, for any application, the number of evaluations required by the *insertion optimisation* and *directive pruning* is constant. *Insertion optimisation* must evaluate all `FOR` loops, and *directive pruning* must evaluate all loops found to be parallelisable by the *insertion optimisation* step. Therefore, given a fixed evaluation budget, after running *insertion optimisation* we know the number of evaluations left to divide between the *data optimisation* and *parameter optimisation* stages. We first calculate what the maximum number of evaluations would be to exhaustively search each genotype in the *data optimisation* and *parameter optimisation* stages. We then weigh the *parameter optimisation*'s 'maximum number of evaluations' figure by 0.5. The reason for this is we observed that the *parameter optimisation* step is of low value compared to the *data optimisation* step. We then split the remaining number of evaluations (after taking into account that used/to be used by the *insertion optimisation* and *data debugging* stages) between the *data optimisation* and *parameter optimisation* stages proportional to their respective 'maximum number of evaluations' value.

2.3 Environment

We compiled the solutions produced by GB-GP-Parallelisation and Four-Stage-Parallelisation using the PGI 16.7 compiler. The PGI 16.7 compiler, provided by The Portland Group, compiles the OpenACC annotated source code under the OpenACC 2.5 standard. We ran the experiments on an Ubuntu 14.04.5 LTS Desktop system with an Intel Core i5-650 processor (3.2 GHz, 2 cores), 4GB of RAM and an nVidia GeForce GTX 1060 GPU. Genetic improvement implicitly optimises for the target hardware. Therefore any optimal solutions found in this investigation may not be optimal across all hardware targets [10]. We believe this to be an advantage of our approach rather than a hinderance. It allows software to be optimised in respect to the deployment environment. In some regards this work echoes previous work on using automatic parameter tuning to optimise kernels for specific GPU setups [14, 16]

2.4 Application Selection and profiling

In this work we targeted the Seoul National University NAS Parallel Benchmark (SNU-NPB) Suite [6], version 1.0.3. This benchmark suite is derived from the NAS Parallel Benchmark (NPB) suite [4], a collection of applications designed to benchmark parallel super computers. The NPB suite is written mostly in FORTRAN which neither of our approaches, at present, can process. The SNU-NPB variant has translated the suite to the C programming language and includes a sequential version of the applications, which we target in this investigation. The suite contains seven applications, the details of which are outlined in Table 1.

Application	Description	# C Files (Targeted)	LOC (Targeted)	Input Class	Execution Time (s)
BT	**B**lock **T**ridiagonal: Solves a synthetic system of non-linear partial differential equations using block tridiagonal matrices.	14 (6)	2,456 (1,419)	Custom*	14.39
CG	**C**onjugate **G**radiant: Using a conjugate gradient method, estimates the smallest eigenvalue of a large sparse symmetric positive-definite matrix.	2 (1)	491 (262)	Custom*	7.91
EP	**E**mbarrassingly **P**arallel: Generates independent Gaussian random variates using an 'embarrassingly parallel' approach.	4 (2)	365 (182)	A	25.92
FT	**F**ourier **T**ransform: Solves a three-dimensional partial differential equation using a fast Fourier transform.	6 (3)	613 (224)	A	7.93
LU	**L**ower-**U**pper: Solves a synthetic system of non-linear partial differential equations using a a lower and upper triangular matrix.	18 (6)	2,183 (1,053)	W	5.48
MG	**M**ulti**G**rid: Estimates the solution of a 3-dimensional discrete Poisson equation using the V-cycle multi-grid method.	2 (2)	878 (387)	B	7.45
SP	**S**calar **P**entadiagonal: Solves a synthetic system of non-linear partial differential equations using a scaler pentadiagonal matrix.	17 (5)	1,944 (1,065)	W	5.23
TOTAL	—	63 (25)	8,930 (4,592)	—	74.31
AVERAGE	—	9.0 (3.6)	1,275.7 (656.0)	—	10.62

Table 1: The sequential applications within the NAS-NPB Suite, showing the number of C files, line of code, input class used, and execution time. Files and lines targeted by our approaches are in parenthesis. *Custom input classes defined in text.

The applications in the SNU-NPB suite each have a selection of input classes (input data with corresponding output data; essentially black box tests). When optimising the applications we use a single input class to train and evaluate on. For each application we selected the input class that ran for more than 5 seconds (so that smaller reductions in execution time could be detectable and not confused with statistical variance) but less than 30 seconds (to keep evaluation times at a manageable level). If two or more input classes fell within this range the one with the lowest execution time was chosen. If there were no input classes a custom input class was created.

Both BT and CG required custom classes to be created. A problem class for BT was created with 40x40x40 grids over 200 times steps with DT equal to 0.8×10^{-3}. For CG, we created a problem class with a size of 30,000 over 30 iterations.

For each application we only target a subset of the code for optimisation. To determine what subset to use we profiled each application using GPROF 2.24 (with the code compiled using GCC 4.8.4) running each application's respective input class. We focused on the level of C functions and selected the minimum set of functions that accounted for over 90% of execution time. We also included the applications' main methods as, in a manner of speaking, this accounts for 100% of execution time. The details of how many files/LOC were ultimately selected from this process is noted in Table 1 (in parenthesis).

Our reasoning for selecting the SNU-NPB suite above others is its applications are written in C (thereby compatible with our frameworks), they can be compiled by the PGI 16.7 compiler we use in our experiments, they contain test data so the correctness of program variants can quickly be verified, they run in a non-trivial amount of time (lower execution times make results more susceptible to statistical error), and we know that the applications are parallelisable. One other important factor in our decision is the existence of the SNU-NPB-ACC suite [2, 18] .

The SNU-NPB-ACC suite contains all the applications found within the SNU-NPB sequential suite but manually annotated with OpenACC directives, thereby parallelising them. We treat these hand-implemented OpenACC directives as a 'ground truth' for both GB-GP-Parallelisation and Four-Stage-Parallelisation. That is, a good estimate for what is achievable when targeting these applications. We compare the patches created by our approaches, for each application, against the solutions within the SNU-NPB-ACC suite in order to observe how our approaches may be modified to achieve better parallelisation; to behave more like a human expert.

2.4.1 Fitness Functions

All the applications targeted have a fitness function associated with them. Shown in Algorithm 3 is the template fitness function we use in our investigations. It should be noted, each application has its own fitness function with the application directory D, the execution timeout T, and the input data I hard-coded. This is because, in both our approaches, the fitness function is

expected to accept only one parameter: the patch, P.

Algorithm 3 The skeleton fitness function used for evaluating each application

Require: D, the directory of the application
P, the input patch (optional)
T, execution timeout, $T \in \mathbb{N}$
I, input data
is_present(P), returns True if that P is present, otherwise false
apply_patch(P, D), returns the directory D with patch P applied
compile(D), compiles the code in D, returns True if successful
run(D, I), executes D on input I and returns the output
timeout(T, C), executes C, capping its execution to time T
is_successful(O), returns True of the output O , is valid
execution_time(O), returns the execution time from the output O

```
1: if is_present(P) then
2:    D' ← apply_patch(P, D)
3: else
4:    D' ← D
5: end if
6: R ← −1 #R, the value to be returned
7: if compile(D') then
8:    if timeout(T, O_r ←run(D', I)) then
9:       if is_successful(O_r) then
10:         R ←execution_time(O_r)
11:      end if
12:   end if
13: end if
14: return R
```

In this algorithm we allow the fitness function to evaluate the program without any optimisation. To do this no patch, P, is given. If present, the patch is applied then the application is compiled. If this compilation was unsuccessful, then '-1' is ultimately returned. If compiled successfully we ran the application on its test input and record the program's output O_r. We utilise the GNU project's `timeout` utility (version 8.21) to put an upper-limit on the evaluation time of a solution. The `timeout` utility will produce a non-zero exit code for a timeout, otherwise return the exit code of the application. In Algorithm 3, we abstract this to a simple function which returns true if no timeout occurred. If the application crashes during execution as it relays the application's exit code through the timeout function (meaning the function *timeout* returns false in Algorithm 3). The output of the program, O_r, contains two pieces of information: the execution time, and whether the application ran successfully — that is, whether it produced the correct output for the corresponding input. Using O_r we output the execution time if the application run successfully and '-1' if it did not.

2.5 Methodology

We allocated 600 evaluations, for each application, when running both GB-GP-Parallelisation and Four-Stage-Parallelisation. For GB-GP-Parallelisation we set a population size of 60, over 10 generations, with a crossover and mutation rate of 0.33, and a tournament size of 6 (10%). The allocation of evaluations between the four stages for Four-Stage-Parallelisation is discussed in Section 2.2. For each we set the evaluation timeout (in the fitness function, see Algorithm 3) to 10 minutes.

For each application, we run the GB-GP-Parallelisation solution, the Four-Stage-Parallelisation solution, the original sequential application as found in the SNU-NPB, and the handwritten OpenACC implementation found in SNU-NPB-ACC, with the same input (see Table 1) 100 times, recording the execution time for each run.

With this data we then used the Wilcoxon rank sum test to declare whether GB-GP-Parallelisation or Four-Stage-Parallelisation were capable of reducing execution time by a statistically significant extent ($p < 0.05$) when compared the sequential source. We also noted the mean percentage change in execution time when comparing the solutions produced by GB-GP-Parallelisation and Four-Stage-Parallelisation compared to the sequential and handwritten OpenACC implementations for each application.

3 Results

In this section we show the results of carrying out the methodology outlined in Section 2.5, using the GB-SP-Parallelisation and Four-Stage-Parallelisation techniques described in Sections 2.1 and 2.2, to answer the research questions outlined in the paper's introduction.

3.1 GB-GP-Parallelisation

To answer RQ1a, *What execution time reductions are achievable when using GB-GP-Parallelisation?*, we produced Figure 5. This bar chart shows the performance of GB-GP-Parallelisation; the optimal applications' execution time is shown as a percentage of the original, unmodified, sequential equivalent. We found that GB-GP-Parallelisation reduces the execution time of CG, EP, and SP by 2.64%, 15.56%, 0.25%, on average, respectively. For all other applications no statistically significant change in execution time was observed. Treating those with no statistically significant change as a 0% reduction, we can say GB-GP-Parallelisation reduces execution time, across all applications, by 2.79% on average.

In answering RQ1b, *How do the reductions in execution time compare to handwritten OpenACC implementations?*, we produced Figure 6. This bar chart shows how these solutions compare to handwritten OpenACC implementations

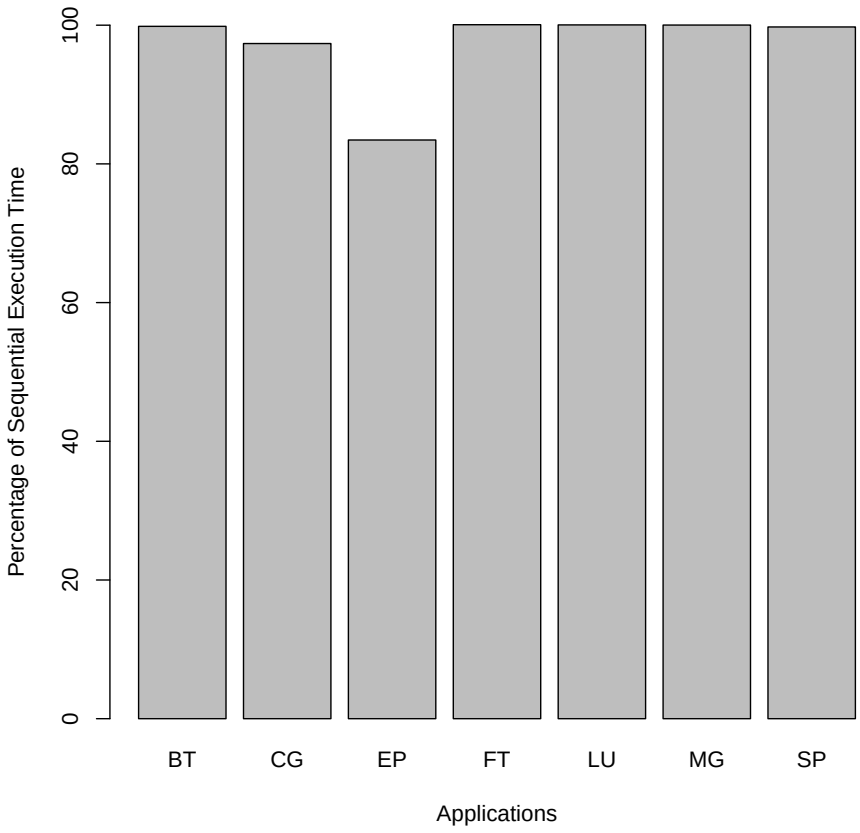


Figure 5: The performance of application optimisations using GB-GP-Parallelisation

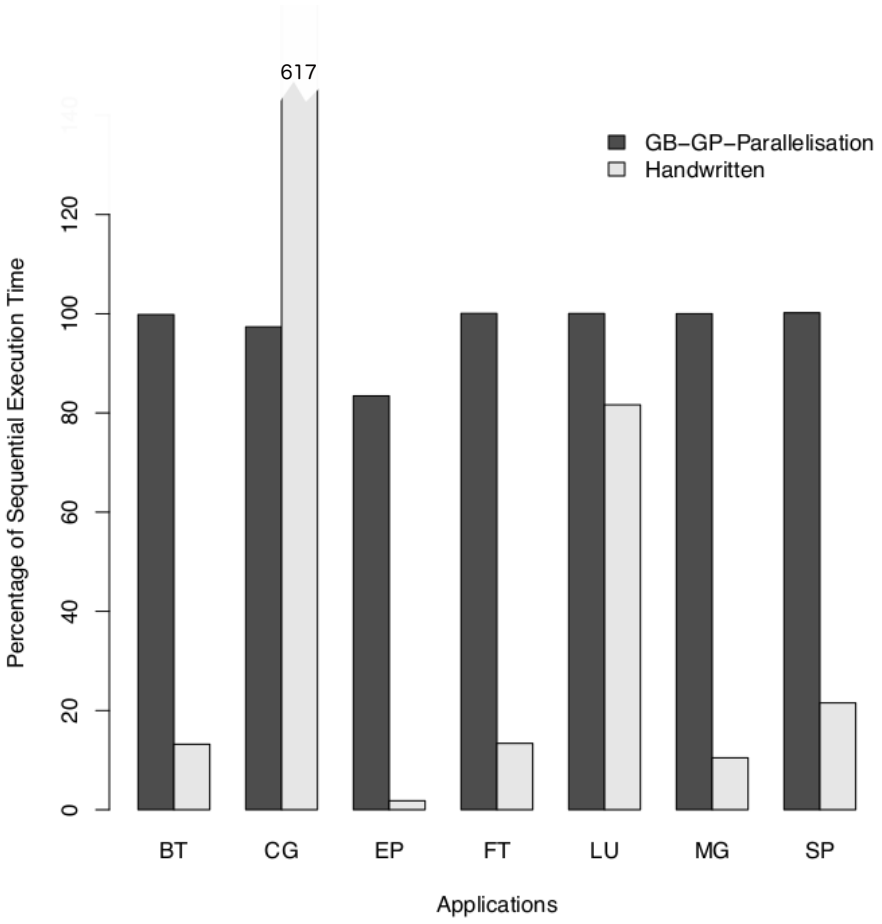


Figure 6: The performance of application optimisations using GB-GP-Parallelisation against the handwritten OpenACC implementation

found in the SNU-NPB-ACC suite. As can be seen, only in one case does GB-GP-Parallelisation improve upon that produced by a human implementation. This one case is not due to the optimisation reducing execution time by a significant extent (the CG solution only reduces execution time by 2.64%). It is because the handwritten OpenACC implementation for CG increases execution time by 517%. We do not know why the handwritten CG implementation runs so poorly on our setup though we suspect this is due to some OpenACC optimisations simply not being universally beneficial across all hardware targets.

Treating CG as a 0% reduction, the handwritten OpenACC implementation reduces execution time by an average of 65.68%. It is therefore evident that GB-GP-Parallelisation could be improved upon significantly if we take the handwritten implementation as a guide to what is achievable.

3.2 Four-Stage-Parallelisation

In answering RQ2a, *What execution time reductions are achievable when using Four-Stage-Parallelisation?*, we produced the bar chart shown in Figure 7. Compared to that produced by GB-GP-Parallelisation, they are similar. A statistically significant decrease in execution time was only found in one case, EP, with a reduction of 17.09%. Due to how Four-Stage-Parallelisation works, it is possible for the 'optimal' solution produced to increase execution time. This is due to the *directive pruning* stage, which utilises a greedy approach to selecting the optimal subset of loop parallelisations. In the case of LU, the optimal solution solution found with no modifications to the source code (i.e. the 'delta debugging' stage removed all the inserted directives). In both Figure 7 and 8, we set LU as having an execution time of 100% of the sequential. Bar LU, all the effects found were statistically significant. BT increased execution time by 1.37%, CG 0.31%, FT 1.02%, MG 0.10%, and SP 13.50%.

If we consider an increase in execution time to be a 0% decrease (as any user of Four-Stage-Parallelisation would choose the original sequential application in the case where it increased execution), we say Four-Stage-Parallelisation decreases execution time by 2.44% on average across all applications. This is slightly lower than GB-GP-Parallelisation's 2.79% reduction but the difference between the two is minimal. Given Four-Stage-Parallelisation's tendency to increase execution time, and the slightly lower reduction on average, GB-GP-Parallelisation appears superior to Four-Stage-Parallelisation. However, both are inferior when compared to the handwritten, OpenACC implementation.

In answering RQ2b, *How do the reductions in execution time compare to handwritten OpenACC implementations?*, we produced Figure 8. This bar chart shows how the optimisations produced by Four-Stage-Parallelisation compare against the handwritten OpenACC implementation. As in the case of GB-GP-Parallelisation, there is only one case in which a solution produced by Four-Stage-Parallelisation is superior, that of CG. Again this is due to the very poor performance of CG and the solution found by Four-Stage-Parallelisation for CG actually increases by 0.31%.

Just as in the case of GB-GP-Parallelisation, we can see there are significant

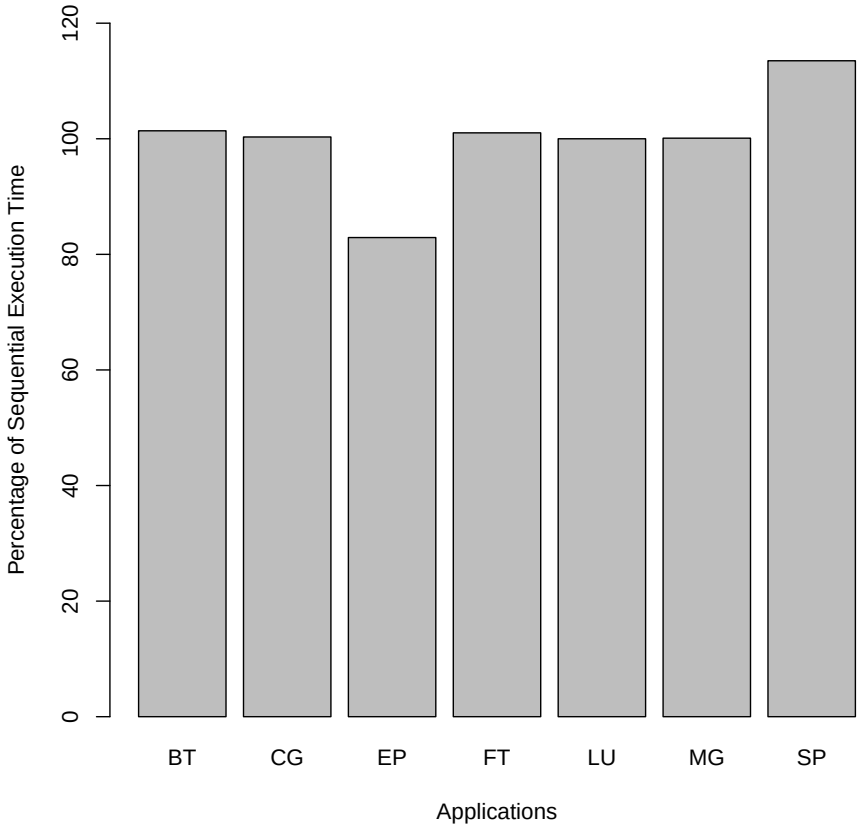


Figure 7: The performance of application optimisations using Four-Stage-Parallelisation

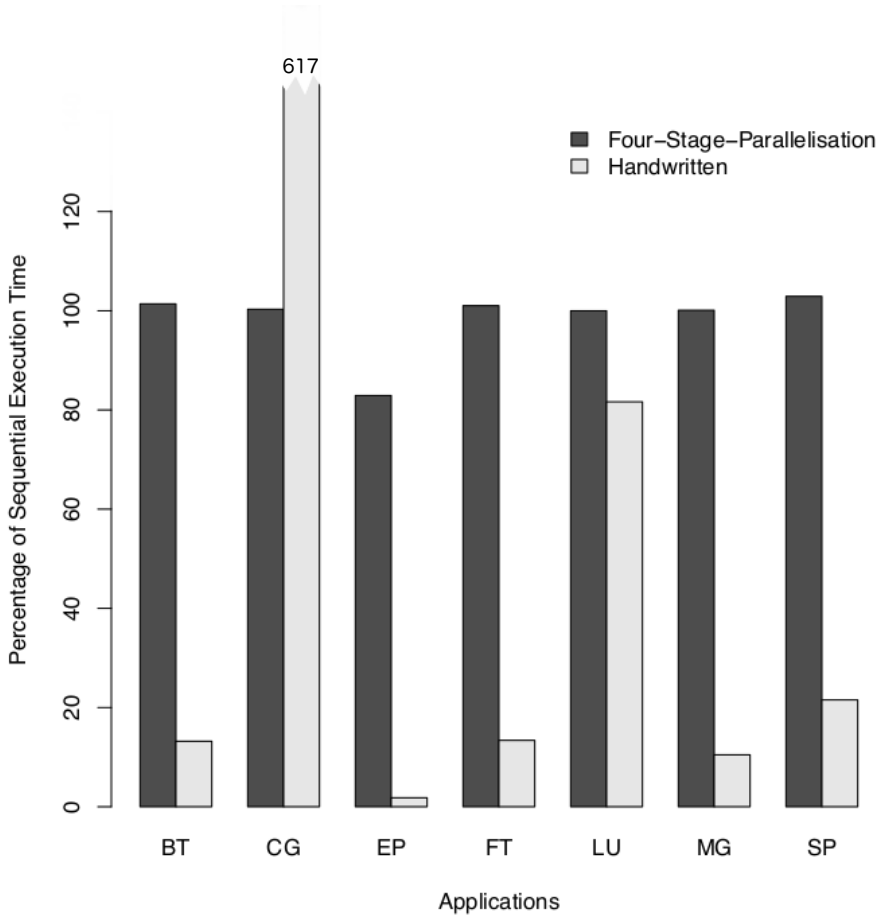


Figure 8: The performance of application optimisations using Four-Stage-Parallelisation against the handwritten OpenACC implementation

performance gains that are not being exploited. The question therefore stands, what does the handwritten implementation do that GB-GP-Parallelisation and Four-Stage-Parallelisation do not?

3.3 Comparison to handwritten OpenACC

We compared the best solution for each application, for both the automatic parallelisation approaches, against the handwritten solutions in the SNU-NPB-ACC suite. In doing so we answered RQ3, *What differs between the solutions produced by GB-GP-Parallelisation, Four-Stage-Parallelisation, and the handwritten OpenACC implementations?* We attempt to explain what the handwritten solutions do which the solutions produced by the techniques outlined in this paper do not.

Upon comparing the solutions we find that the most significant difference is the SNU-NPB-ACC suite's use of OpenACC's `#pragma acc data` directive. These directives create what are known as 'data regions' — scopes of code for which data created or copied into the GPU will continue to exist until the scope is completed. For example, `#pragma acc data create(x){ <C code scope> }` will create the variable `x` on the GPU and it will exist on the GPU for the entirety of the `<C code scope>` segment. Both GB-GP-Parallelisation and Four-Stage-Parallelisation place emphasis on the insertion of `#pragma acc parallel loop` directives which created an implicit data region for the body of the *FOR* loop it parallelises. However, we find (when looking at the handwritten OpenACC implementations) that significant efficiencies can be gained when ensuring many parallelised loops exist in the same data region. We also find the SNU-NPB-ACC suite makes use of `acc_malloc`, a special function in OpenACC with similar functionality as the standard C library's `malloc` function except it allocates memory on the hardware accelerator (in this case the GPU) instead of mean memory. This is another form of data optimisation.

In all the applications within the SNU-NPB-ACC suite we find that all the loop parallelisations are contained within a single data region. The role of this common data region is to create commonly utilised variables on the GPU at the beginning of the programs execution. Then, within this grand data region, the parallelised `FOR` loops declare that the data they need is already within the GPU and, therefore, it is not necessary to carry out the costly operation of copying from main computer memory or copying back when the loop has finished processing.

Figure 9 shows some (abridged) source code from the FT application in the SNU-NPB suite to demonstrate this. The `main` method contains a data directive which covers two methods, `init_ui` and `evolve`. This data directive 'creates' 11 arrays on the GPU (these are declared as global variables elsewhere in the program). Then, whenever a loop is parallelised and requires one of these variables, the `present` argument is used, as in `unit_ui` (line 20). This OpenACC directive informs the compiler that `u0_real`, `u0_imag`, `u1_real`, `u1_imag`, and `twiddle` are present on the GPU and, therefore, do not need copied over from maim memory prior to processing or back to main memory after.

App	GB-GP-Parallelisation	Four-Stage-Parallelisation	Handwritten
BT	0	1	44
CG	1	10	8
EP	3	2	5
FT	0	1	12
LU	0	0	59
MG	1	1	24
SP	0	2	65

Table 2: The number of FOR loop structures parallelised (for nested loops, only the outermost parallelised loop is counted)

With such a setup it is important to copy the data back to main memory when it's needed and update the value in the GPU if changed in main memory (or vice-versa). OpenACC uses the terminology 'host' to refer to the systems main memory and 'device' to refer to the targeted hardware accelerator's memory, in this case, that of the GPU. This is the role of the `#pragma acc update host(<variable>)` and `#pragma acc update device(<variable>)` directives.

We find that, in the applications studied, many loops are parallelisable. However, simply adding a `#pragma acc parallel loop` directive is insufficient. For each `FOR` loop there is an overhead where data must be transferred to the GPU (device) memory before processing and back to the main (host) memory after. Frequently, this makes the parallelised version even more costly than the sequential. However, if these fixed overheads can be shared over multiple for-loop parallelisations the cost can be reduced. When the `#pragma acc data` directive is used effectively the transfer to and from the GPU can be kept at a minimum and it is this, we find, is when significant gains can be made.

Both our approaches attempt to parallelise `FOR` loops, toggle their parameters, and optimise data flow solely within the `FOR` loop. It is possible in GB-GP-Parallelisation for a data directive to be created (it is contained within the grammar) but the search space is vast. Only in one instance, EP, did we find a data region being implemented, though this data region did not specify any variables and covered 3 statements which were not parallelised. This data region was, therefore, inert. It is clear comparing the solutions found via our approach, that we must encompass these `FOR` loop parallelisations into larger data regions.

We find that due to the lack of proper mechanisms to create meaningful data directives, neither approach parallelises many `FOR` loop structures. Table 2 shows the number of `FOR` loops champion parallelised solution for GB-GP-Parallelisation, Four-Stage-Parallelisation, and the handwritten version (for cases of nested `FOR` loops, we only count the outer-most parallelisation). As can be seen, the handwritten implementation frequently parallelises more `FOR` loops than either GB-GP-Parallelisation and Four-Stage-Parallelisation.

To emphasise what a dramatic effect correct use of data directives can have, we took the handwritten version and stripped it from OpenACC data directives (`#pragma acc data ...`), update statement (`#pragma acc update`

App	Handwritten (s)	Sequential (s)	Handwritten without data directives (s)
BT	1.90	14.40	3463.23
CG	48.80	7.91	11.74
EP	0.47	25.92	48.11
FT	1.06	7.93	34.64
LU	4.47	5.23	970.11
MG	0.78	5.48	N/A (see text)
SP	1.24	5.48	1484.74

Table 3: The handwritten OpenACC implementation's execution time compared to its variant without data directives (timeout is 2 hours).

`host/device`), data parameters on the `FOR` loop declarations (i.e. `copy`, `copyin`, `copyout`, `create`, and `present`), and replaced instances of `acc_malloc` with `malloc`. In cases where removal of these data directives resulted in a `FOR` loop parallelisation being uncompilable we manually removed the parallelisation. We recorded the execution time prior and after this change. These times can be seen in Table 3. Even compared to the execution time of the sequential time, parallelising loops without correctly applying data directives results in significantly increased execution time (in the case of MG, removing the data directives resulted in an error at execution time).

4 Discussion

Both GB-GP-Parallelisation and Four-Stage-Parallelisation employed different approaches to automatic optimisation. Both reduced execution time by less than 3%, 2.79% for GB-GP-Parallelisation and 2.44% for GB-GP-Parallelisation (averaged over all applications). We consider GB-GP-Parallelisation to be superior compared to Four-Stage-Parallelisation, not only in that it reduces execution time by a greater extend, but also that it does not produce any solutions that are worse than the original, which Four-Stage-Parallelisation does in five of the seven applications targeted. However, solutions that run in a higher execution time can easily be reverted back to the original, so GB-GP-Parallelisation's superiority is only slight.

In both instances the EP application shows the biggest reduction in execution time with GB-GP-Parallelisation achieving a 16.56% reduction and Four-Stage-Parallelisation achieving a 17.09% reduction. This is, perhaps, no surprise given this application given its full name — 'embarrassingly parallel'. However, even in this case, we found we could not match the handwritten OpenACC implementation's performance. This handwritten OpenACC implementation is capable of reducing EP's execution time is 98.19%. The goal of this research was not to beat, or even match, that capable of a human expert (though such results would have been welcome). Rather we wished to develop a technique that could meaningfully parallelise software automatically. Just as a compiler

will rarely produce the optimal machine code implementation for a given source, we do not envision a future in which genetic improvement is unbeatable, but one in which genetic improvement is 'good enough' — so cheap and easy to use that it can justify its existence on economic grounds. It is difficult to know at what point genetic improvement can begin to replace human effort, and will undoubtably differ on case-by-case basis. However, in this case, the optimisations produced by GB-GP-Parallelisation and Four-Stage-Parallelisation are lower than we would have hoped and below what can be achieved with a skilled human expert.

Our overarching idea in both the approaches to automatic parallelisation was to focus on parallelising `FOR` loops, optimising the data flow in and out of the loops, and tweaking their parameters. We have found this is too basic an idea and a more holistic one is required. We have found, when comparing to that produced by human experts, `FOR` loops share the same data and managing this data in a common way can significantly reduce the overheads of parallelisation. Moving data to and from the GPU is costly and this cost can (and in our experience, often does) destroy the gains of parallelisation. We believe future research should focus on this problem. It is one in which search-based techniques are well equipped. The objective is to optimise at what points data is transferred to and from main memory. This would not need to be a black-box optimisation as the source code reveals where data is used. For example, in the case that two parallelised `FOR` loops, one after another, use the same variables, it is logical that these `FOR` loops should share the same data region as data moved to the GPU for the first may as well continue to stay there for the second.

Once research has solved this problem we believe that we will begin to see significant execution time reductions for the applications studied. As is evident, the NAS Parallel benchmark suite studied consists of applications we know can be parallelised. We do not see this as a fault in our investigation. This is a new approach and therefore needs an easy example to try first. Just as we learn to walk before we learn to run, we target these benchmarks before moving into more challenging ones. In the medium term, once we have successfully optimised these benchmarks, we would like to test our techniques on 'real world' applications in which there exists no parallelised equivalent to see what can be achieved, and what other research considerations should be taken into account. If we can achieve noteworthy results in this domain then the software engineering community can begin to reap the significant benefits of automatic parallelisation.

5 Conclusion

We have developed two approaches to automatic parallelisation of sequential software: GB-GP-Parallelisation and Four-Stage-Parallelisation. These approaches create and insert OpenACC directives into C programs with the goal of decreasing execution time by offloading activity to the system's GPU. GB-GP-Parallelisation uses grammar-based GP, while Four-Stage-Parallelisation uses both greedy algorithms and evolutionary strategies across four separate stages

of optimisation. When attempting to parallelise a sequential implementation of the NAS Parallel Benchmark suite, we find that GB-GP-Parallelisation reduces execution time by 2.79% and Four-Stage-Parallelisation by 2.44% on average. We compare this to the NAS Parallel Benchmark suite with hand-implemented OpenACC directives. The hand-implemented variant reduces execution time by 65.68% on average, showing that both GB-GP-Parallelisation and Four-Stage-Parallelisation fall short of producing the substantial decreases in execution time we know is possible. We carried out a comparison between the solutions produces between GB-GP-Parallelisation, Four-Stage-Parallelisation, and the handwritten OpenACC variant and found GB-GP-Parallelisation and Four-Stage-Parallelisation both under-perform due to their poor handling of how program variables are transferred to and from the GPU during program execution. We would therefore advise future researchers of automatic parallelisation methods to focus their efforts on how to automatically optimise data transfer to and from the GPU.

Acknowledgements

We wish to thank Prof. William B. Langdon, Prof. Mark Harman, and Dr. Earl T. Barr for their help in carrying out this investigation.

References

[1] DawnCC — A source-to-source compiler for parallelizing C/C++ programs with code annotations. `https://github.com/gleisonsdm/DawnCC-Compiler`. Accessed: 7-November-2017.

[2] NAS_OpenACC_2.5. `https://github.com/spino327/NAS_OpenACC_2.5`. Accessed: 3-August-2017.

[3] Thomas Back, Frank Hoffmeister, and Hans-Paul Schwefel. A survey of evolutionary strategies. In *Proceedings of the 1991 International Conference on Genetic Algorithms*, volume 2. Morgan Kaufmann Publishers, San Mateo, 1991.

[4] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Rob S. Schreiber, et al. The NAS parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[5] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming: An introduction*, volume 1. Morgan Kaufmann, San Francisco, 1998.

[6] Center for Manycore Programming. SNU NPB suite. `http://http://aces.snu.ac.kr/software/snu-npb`.

[7] Leonardo Dagum and Ramesh Menon. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[8] Alastair Donaldson, Colin Riley, Anton Lokhmotov, and Andrew Cook. Auto-parallelisation of Sieve C++ programs. In *Proceedings of the 2007 European Conference on Parallel Processing — EuroPar '07*, pages 18–27. Springer, 2007.

[9] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001.

[10] William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3d medical image registration cuda software with genetic programming. In *Proceedings of the 2014 Genetic and Evolutionary Computation Conference — GECCO '14*, pages 951–958. ACM, 2014.

[11] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. *VHDL: Hardware description and design*. Springer Science & Business Media, 2012.

[12] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. DawnCC: automatic annotation for data parallelism and offloading. *ACM Transactions on Architecture and Code Optimization*, 14(2):13, 2017.

[13] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[14] Cedric Nugteren. CLBlast: A tuned OpenCL BLAS library. *arXiv preprint arXiv:1705.05249*, 2017.

[15] Fernando Otero, Tom Castle, and Colin Johnson. Epochx: Genetic programming in Java with statistics and event monitoring. In *Proceedings of the 2012 Annual Conference Companion on Genetic and Evolutionary Computation — GECCO '12*, pages 93–100. ACM, 2012.

[16] Edoardo Paone, Francesco Robino, Gianluca Palermo, Vittorio Zaccaria, Ingo Sander, and Cristina Silvano. Customization of OpenCL applications for efficient task mapping under heterogeneous platform constraints. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition — DATE '15*, pages 736–741. IEEE, 2015.

[17] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. Genetic Improvement of software: A comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 2017.

[18] Sergio Pino, Lori Pollock, and Sunita Chandrasekaran. Exploring translation of OpenMP to OpenACC 2.5: Lessons learned. In *Proceedings of the 7th International Workshop on Accelerators and Hybrid Exascale Systems — AsHES '17*. IEEE, 2017.

[19] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Computing in science & engineering*, 12(3):66–73, 2010.

[20] Peter A. Whigham. Grammatically-based genetic programming. In *Proceedings of the workshop on Genetic Programming: from theory to real-world applications*, volume 16, pages 33–41, 1995.

[21] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC — First experiences with real-world applications. In *Proceedings of the 2012 European Conference on Parallel Processing — EuroPar '12*, pages 859–870. Springer, 2012.

A OpenACC Grammar

OpenACC Grammar

```
<start>::= <start> <start> | <base>
<base>::= "#pragma acc " <choice>
<choice>::= "parallel "  <parallel> <private>
                <reduction> <optional_block> <line_number>
| "loop " <loop> <private> <reduction> <loop_line_number>
| "parallel loop " <loop> <parallel> <private> <reduction>
                <loop_line_number>
| "kernels " <kernels> <private> <block> <line_number>
| "kernels loop " <loop> <kernels> <private>
                <loop_line_number>
| "data " <data> <block> <line_number>
| "cache(" <variables> ") "  <top_loop_line_number>
| "atomic " <atomic_clause> <line_number>
| "update " <async> <wait> <update> <line_number>
| "routine " <routine> <function_line_number>
| "wait " <async> <line_number>
| "wait(" <sync_number> ") " <async> <line_number>
<parallel>::= <async> <wait> <num_gangs> <num_workers>
                 <vector_length> <data> <firstprivate>
<loop>:: = <gang> <worker> <vector> <seq> <collapse>
                <auto> <independent>
<kernels>::= <async> <wait> <data>
<data>::= <copy> <copyin> <copyout> <create> <present>
                <present_or_copy> <present_or_copyin>
                <present_or_copyout> <present_or_create>
<routine>::= <gang_singular><worker_singular> <vector_singular>
                <seq>
<update>::= "self(" <variables> ") "
| "host(" <variables> ") "
| "device(" <variables> ") "
| " "
<async>::= "async "
| "async(" <sync_number> ")"
| " "
<wait>::= "wait "
| "wait(" <sync_number> ")"
| " "
<sync_number>::= "1" | "2" | "3" | "4" | "5"
<num_gangs>::= "num_gangs(" <two_power> ") "
| " "
<num_workers>::= "num_workers(" <two_power> ") "
| " "
```

```
<vector_length>::= "vector_length(" <two_power> ") "
| " "
<reduction>::= "reduction(" <reduction_operator> ":" <variables> ")"
| " "
<reduction_operator>::= "+" | "*" | "max" | "min" | "&"
                        | "|" | "^" | "&&" | "||"
<copy>::= "copy(" <variables> ") "
| " "
<copyin>::= "copyin(" <variables>") "
| " "
<copyout>::= "copyout(" <variables> ") "
| " "
<create>::= "create(" <variables> ") "
| " "
<present>::= "present(" <variables> ") "
| " "
<present_or_copy>::= "present_or_copy(" <variables> ") "
| " "
<present_or_copyin>::= "present_or_copyin(" <variables> ") "
| " "
<present_or_copyout>::= "present_or_copyout(" <variables> ") "
| " "
<present_or_create>::= "present_or_create(" <variables> ") "
| " "
<private>::= "private(" <variables> ") "
| " "
<firstprivate>::= "firstprivate(" <variables> ") "
| " "
<collapse>::= "collapse(" <collapse_number> ") "
| " "
<collapse_number>::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
                        | "9" | "10"
<gang>::= <gang_singular>
| "gang(" <two_power> ") "
<gang_singular>::= "gang " | " "
<worker>::= <worker_singular>
| "worker(" <two_power> ") "
<worker_singular>::= "worker " | " "
<vector>::= <vector_singular>
| "vector(" <two_power> ") "
<vector_singular>::= "vector " | " "
<seq>::= "seq "
| " "
<auto>::= "auto "
| " "
<independent>::= "independent "
```

```
| " "
<atomic_clause>::= "read "
| "write "
| "capture "
| "update " <optional_block>
<optional_block>::= <block>
| " "
<two_power>::= "2" | "4" | "8" | "16" | "32" | "64" | "128" | "256"
                 | "512" | "1024"
<block>::= "\n{ " <block_placeholder>
<variables>::= <variable>
| <variable> "," <variables>
<variable>::= <variable_placeholder>
<block_placeholder>::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
| "9" | "10" | "11" | "12" | "13" | "14" | "15" | "16" | "17" | "18"
| "19" | "20" | "21" | "22" | "23" | "24" | "25" | "26" | "27" | "28"
| "29" | "30" | "31" | "32" | "33" | "34" | "35" | "36" | "37" | "38"
| "39" | "40" | "41" | "42" | "43" | "44" | "45" | "46" | "47" | "48"
| "49" | "50" | "51" | "52" | "53" | "54" | "55" | "56" | "57" | "58"
| "59" | "60" | "61" | "62" | "63" | "64" | "65" | "66" | "67" | "68"
| "69" | "70" | "71" | "72" | "73" | "74" | "75" | "76" | "77" | "78"
| "79" | "80" | "81" | "82" | "83" | "84" | "85" | "86" | "87" | "88"
| "89" | "90" | "91" | "92" | "93" | "94" | "95" | "96" | "97" | "98"
| "99" | "100"
<variable_placeholder>::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
| "9" | "10" | "11" | "12" | "13" | "14" | "15" | "16" | "17" | "18"
| "19" | "20" | "21" | "22" | "23" | "24" | "25" | "26" | "27" | "28"
| "29" | "30" | "31" | "32" | "33" | "34" | "35" | "36" | "37" | "38"
| "39" | "40" | "41" | "42" | "43" | "44" | "45" | "46" | "47" | "48"
| "49" | "50" | "51" | "52" | "53" | "54" | "55" | "56" | "57" | "58"
| "59" | "60" | "61" | "62" | "63" | "64" | "65" | "66" | "67" | "68"
| "69" | "70" | "71" | "72" | "73" | "74" | "75" | "76" | "77" | "78"
| "79" | "80" | "81" | "82" | "83" | "84" | "85" | "86" | "87" | "88"
| "89" | "90" | "91" | "92" | "93" | "94" | "95" | "96" | "97" | "98"
| "99" | "100"
```

App	Input class
BT	Custom*
CG	Custom*
EP	A
FT	A
LU	W
MG	B
SP	W

Table 4: The SNU-NPB applications targeted and the testcase used for evaluation. *Custom input classes defined in text.

B DawnCC Investigation

DawnCC is a compiler module which automatically detects parallelisable code in C/C++ programs and then inserts OpenACC or OpenMP directives where appropriate [12]. We carried out a small investigation to determine DawnCC's performance when inserting OpenACC directives on the Seoul National University NAS Parallel Benchmark suite (SNU-NPB) [6]. The SNU-NPB suite contains sequential versions of seven applications known to be parallelisable using OpenACC [18].

B.1 Experiment setup

Our experiment to evaluate DawnCC's performance was quite simple. For each application within the SNU-NPB suite we ran DawnCC [3] to produce a variant which contained OpenACC directives. We ran both the sequential and DawnCC variant of each application 100 times, on a test case (selection discussed below). We then compared the means of these runs, and whether they were statistically significant (using the Wilocoxon Rank Sum test) to determine DawnCC's effectiveness.

Table 4 shows the apps from the SNU-NPB suite and the test cases we used to evaluate DawnCC against for each application. The applications in the SNU-NPB suite each have a set of input classes (input data with corresponding output data; essentially a black box test). To evaluate the original and the DawnCC variant, we selected the input class that ran in greater that 5 seconds (so that smaller reductions in execution time could be detectable and not confused with statistical variance) and less than 30 seconds (in the interests of keeping experiment times low). If two or more input classes fell within this range then the one with the lowest execution time was chosen. If there were no input classes a custom input class was created.

Both BT and CG required custom classes to be created. A problem class for BT was created with 40x40x40 grids over 200 time steps with DT equal

[3] As available from the DawnCC GitHub [1] on the 7th of November 2017.

App	Sequential Mean (s)	DawnCC Mean (s)	Wilcoxon Rank Sum Test p-value
BT	15.35	73.20	$\ll 0.001$
CG	8.43	11.26	$\ll 0.001$
EP	21.92	27.65	$\ll 0.001$
FT	8.54	8.51	0.670
LU	5.77	8.76	$\ll 0.001$
MG	8.06	8.17	0.005
SP	5.69	5.74	0.009

Table 5: DawnCC's performance on the SNU-NPB suite's sequential implementation.

to 0.8×10^{-3}.For CG we setup a problem class with a size of 30,000 over 30 iterations.

Experiments were carried out on an Ubuntu 14.04.5 LTS Desktop system with an Intel Core i5-650 processor (3.2 GHz, 2 cores), 4GB of RAM and an nVidia GeForce GTX 1060 GPU. The code was compiled using the PGI 17.4-0 C compiler.

B.2 Experiment results

Table 5 shows the results from the experiments. We found that in no case did the DawnCC variant decrease execution time. Of the seven applications studied, six are found to increase execution by a statistically significant extent ($p < 0.01$), with the the SP DawnCC variant having no statistically significant influence on execution time.

```
1  int main(int argc, char *argv[]){
2    #pragma acc dta create(u0_real, u0_imag,\\
3          u1_real, u1_imag, u_real, u_imag, twiddle,\\
4          gty1_real, gty1_imag, gty2_real, gty2_imag)
5    {
6      init_ui(dims[0], dims[1], dims[2]);
7      ...
8      for(iter = 1; iter <=niter; iter++){
9        evolve(dims[0], dims[1], dims[2]);
10       ...
11     }
12     ...
13   }
14   ...
15 }
16
17
18 static void init_ui(int d1, int d2, int d3){
19   int i, j, k;
20 #pragma acc parallel loop num_gangs(d3) num_workers(8)\\
21         vector_length(128) present(u0_real, u0_imag,\\
22         u1_real, u1_imag, twiddle)
23   for(k=0; k < d3; k++){
24     for(j=0; j < d2; j++){
25       for(i=0; i < d1; i++){
26         u0_real[k*d2*(d1+1) + j*(d1+1) + i] = 0.0;
27         u0_imag[k*d2*(d1+1) + j*(d1+1) + i] = 0.0;
28         u1_real[k*d2*(d1+1) + j*(d1+1) + i] = 0.0;
29         u1_imag[k*d2*(d1+1) + j*(d1+1) + i] = 0.0;
30         twiddle[k*d2*(d1+1) + j*(d1+1) + i] = 0.0;
31       }
32     }
33   }
34 }
35
36 static void evolve(int d1, int d2, int d3){
37   int i, j, k;
38   #pragma acc parallel loop present(u_real, u0_imag,\\
39         u1_real, u1_imag, twiddle)
40   for(k=0; k < d3; k++){
41     for(j=0; j < d2; j++){
42       u0_real[k*d2*(d1+1) + j*(d1+1) + i] = \\
43                 u0_real[k*d2*(d1+1) + j*(d1+1) + i] \\
44                 *twiddle[k*d2*(d1+1) + j*(d1+1) + i];
45       u0_imag[k*d2*(d1+1) + j*(d1+1) + i] = \\
46                 u0_imag[k*d2*(d1+1) + j*(d1+1) + i] \\
47                 *twiddle[k*d2*(d1+1) + j*(d1+1) + i];
48       u1_real[k*d2*(d1+1) + j*(d1+1) + i] = \\
49                 u0_real[k*d2*(d1+1) + j*(d1+1) + i];
50       u1_imag[k*d2*(d1+1) + j*(d1+1) + i] = \\
51                 u0_imag[k*d2*(d1+1) + j*(d1+1) + i];
52     }
53   }
54 }
```

Figure 9: Example usage of OpenACC data directive