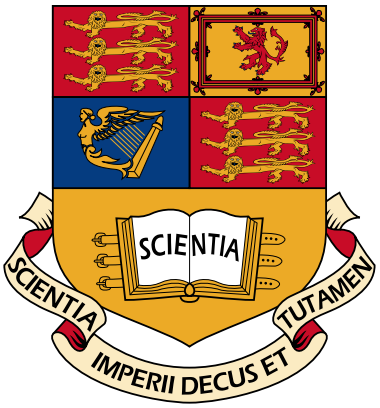
Optimising Unstructured Mesh Computational Fluid Dynamics Applications on Multicores via Machine Learning and Code Transformation

Roxana Rusitoru (rr908)
Supervisor: Prof. Paul Kelly
Second marker: Dr. Tony Field



Department of Computing
Imperial College London

June 19, 2012

Abstract

We show that case-based reasoning (CBR) and deterministic code analysis can be successfully used in optimizing compilers of unstructured mesh applications to obtain better execution times. With the recent shift of CPU architectures towards SIMD capabilities, and of GPU architectures towards general purpose computing, it is no longer clear what optimizations are optimal given a particular problem and target architecture. As a result, we explore the use of machine learning and deterministic algorithms on OP2 C++ Airfoil variations to determine whether such methods can provide optimal or near-optimal results. Our choice of optimizations are loop fusion and runtime parameter variation (block size, partition size and warpsize).

The new perspectives we are exploring in this project are determining optimisations by looking at OP2 code and user kernel complexity, irrespective of low-level architecture details, the integration of a CBR system and deterministic methods to significantly prune our search space and our focus on multiple heterogeneous architectures (CPUs, GPUs).

Acknowledgements

I would like to thank my supervisors, Prof. Paul Kelly and Dr. Tony Field, and Dr. Carlo Bertolli for the great support and guidance they have given me throughout this project. I would also express my deepest gratitude for all the inspiring conversations we had and all the great ideas we shared.

My great thanks to Dr. Nicolas Loriant for all his advice and for his loop fusion implementation into the OP2 compiler, and Dr. Adam Betts for his help and initial implementation of the OP2 to OpenCL compiler.

Finally, I would like to thank my family and friends for their continuous support throughout the year.

Contents

Chapter 1

Introduction

In this project we investigate machine learning and deterministic techniques within an optimizing compiler. We show what impact such methods have upon the performance of the program, by using case-based reasoning and code analysis to select an optimal set of optimizations from a pool of possibilities. This methodology is scalable and offers great advantages over brute-force approaches, as it significantly reduces overall execution time whilst determining the optimal set of optimizations. To achieve this, we used OP2 and its associated compiler, as a framework for the machine learning techniques.

1.1 Motivation

Recent developments in Central Processing Unit (CPU) architectures have moved focus from a serial to a parallel program execution [1]. This change was mostly driven by increasing power consumption requirements to run fast serial programs. At the same time, in line with an idea of taking advantage of existing parallelism, Graphics Processing Unit (GPU) architectures have become more efficient at running general purpose code on them [2], as opposed to just exclusive graphical workloads. In order to fully utilize the capabilities of the hardware (CPU or GPU), we need to focus our abilities into adapting or writing the software in a parallel manner. This is particularly important in High Performance Computing, an area of computing focused on obtaining maximum performance for a particular piece of software on a specific highly parallelized architecture.

Given these developments, we need to rethink the way we write software so that we can take advantage of multiple parallel processing units. This can generally be achieved by applying a number of parallelization techniques to the application. However, it is not always clear which ones are best

suited. This project explores the usage of a number of such techniques on unstructured mesh applications, by making use of machine learning. The Artificial Intelligence system is intended to aid us in making informed decisions as to which optimization procedure is best. Machine learning is comprised of a class of mechanisms that allow a computer to make informed decisions and present evolving behaviours, much like a human.

Achieving such a goal requires an optimizing framework on top of which we can add our contributions and the actual machine learning mechanism. For the framework, we need a target language which we compile down to, and that can be run across a number of architectures. As a result, we chose to work with OpenCL [3], an open standard and framework, which allows the user to run the same code across different platforms (GPUs, CPUs).

For the machine learning system we investigate a case-based reasoning (CBR) mechanism. Case-based reasoning is an Artificial Intelligence technique that allows a machine to solve problems based on past problems. Its main advantage is that it keeps learning from new experiences and thus allows it to adapt to previously unseen situations. We chose this policy decision technique because it keeps learning even after the initial training phase and adapts to new situations, which is something we expect to encounter.

In order to maintain future flexibility and practicality for our solution, our framework of choice is an existing optimizing framework called OP2 [4]. OP2 is an open-source framework that is used to execute unstructured grid applications on a number of platforms, such as GPUs and CPUs. This allows the user to write one piece of code using the OP2 library, and then the OP2 compiler takes it and generates code that runs on one of a number of back-ends. Currently, the library supports few back-end implementations, including OpenCL. Its main drawback, and our advantage, is that for each given program it only has one output solution. This is given by the OP2 compiler. Therefore, we are going to add the policy decision mechanism in a script that calls the compiler. Based on the machine learning optimization result, we choose appropriate compilation flags. By using OP2 and its compiler, we have a practical advantage of extending on an existing framework that already has applications written for it. It is also beneficial that is saves us implementation time, as there is no need to write a compiler from scratch, thus allowing us to focus on the main aspects of the project.

1.2 Objectives

This project aims to investigate the performance of various optimization techniques on unstructured meshes and the usage of a machine learning

system and code analysis to better choose between these options. This will allow us to better understand the connection between different optimization choices and existing platforms, to the extent that we can offer valuable extensions to similar projects which do not make use of an informed decision making process, when selecting their parallelization techniques. As OP2 is one such framework, we are incorporating all changes in it.

It is important to show the role of machine learning systems and code analysis within the context of software optimization, as alternatives generally assume a brute-force approach when selecting an optimization. However, applications such as industrial-scale ones can take weeks to run, therefore, trying every single possibility is not feasible and an efficient search space pruning method is preferred.

We evaluate the success of the added optimizations, code analysis and machine learning mechanism by comparing new execution times against existing ones, obtained by running code within the OP2 framework. For benchmarking, we use a fluid dynamics toy application called Airfoil [5], which simulates a good number of issues found in industrial-scale applications, such as Hydra. This is a finite volume, unstructured mesh turbomachinery computational fluid dynamics application used in production by Rolls Royce. In the final stages, we offer a future work case study for an application of our results onto Hydra.

1.3 Contributions

With this project, we make the following contributions:

- We adjusted the existing OpenCL OP2 runtime support to match the OP2 standards and ensure its correct execution on both GPUs and CPUs. We also added runtime support for our tuning mechanism.
- We fixed and improved the untested OP2 OpenCL compiler.
- We show the connection between various optimization parameters, such as possible loop fusions, block size, partition size and warpsize and a target architecture by using machine learning techniques and code analysis. We also present a way of determining whether loop fusion is possible or not, given two loops.
- We explore the efficiency of the optimizations and the use of machine learning on a toy fluid dynamics application called Airfoil.

- We discuss the applicability of the presented solution on an industrial-scale application called Hydra.

1.4 Report outline

The remainder of the report has the following outline:

- Chapter 2 contains a background of the notions necessary for a better comprehension of the topic.
- Chapter 3 expands on related work and our research goals.
- Chapter 4 details on prerequisites for the successful utilisation of the OP2 framework and compiler.
- Chapter 5 expands on machine learning design and implementation details.
- Chapter 6 provides a performance evaluation of the techniques presented in Chapter 5.
- Chapter 7 presents a case study for an application of our results onto Hydra.
- Chapter 8 gives a summary of the project's contributions and future work.

Chapter 2

Background

In order to allow for a wider range of comparison possibilities we choose to use a number of architectures and techniques that apply to those. This allowed us to gain a better understanding of the connection between a parallelization mechanism and a particular platform. For the purpose of this, we will summarize all the notions that are needed to better understand the topic and techniques used in this project.

2.1 Architectures

2.1.1 Multi-core

Multi-core architectures [6], traditionally, have the following structure: on a die, they contain a few (typically 2–8) very complex cores. These can run general purpose code without an issue, due to their advanced instruction sets and capabilities. CPUs have, since their conception, been implementing this model. This offers great flexibility with regards to accommodating many types of applications, however it comes at a potential performance cost. CPUs have two types of memory: local (called cache) and global, which is the actual global memory of the system. The purpose of the caches is to store data that is immediately needed by the CPU, as these memories are a lot faster than main memory, thus not halting the CPU. An improper use of those, such as not ever having the right data in the cache, can lead to significant performance reductions. Furthermore due to their relatively low number of cores, there is only so much application parallelism that can be exploited.

Lately, in order to improve the performance of these devices, Single Instruction Multiple Data capabilities have been added to them, similarly to many-core architectures. This means that we can now process multiple

array elements at the same time, instead of just one, assuming we only apply one instruction to all of them. This approach is similar to what GPUs offer, however, there is a significant difference between them, as CPUs allow the execution of 4–8 elements in parallel, whilst GPUs can allow thousands. More information about CPU architectures can be found on Intel or AMD's website.

2.1.1.1 AVX/SSE

Advanced Vector Extensions (AVX) [7] and Streaming SIMD Extensions (SSE) [8] are two sets of extensions to x86 architectures. As the name implies, they are SIMD (Single Instruction, Multiple Data) or vector instructions which allow to parallelize the their execution over independent sets of data, where within each set, there is no dependency between any of the elements). These instructions can be heavily used in applications such the unstructured mesh one due to the fact that most parts of the mesh can be processed in parallel. For example: if the mesh contains triangles as base elements, then we can process in parallel data related to two unrelated triangles. This is possible as we apply the same operations to all elements.

2.1.2 Many-core

Many-core architectures are typically based on many, simple, small cores, thus being able to pack hundreds of them on one chip. These are then grouped into higher-level units, such that all cores in a unit can run the same instruction at the same time on multiple different pieces of data. Their limitation, however, is that they do not offer the same capabilities as multi-core architectures, due to the reduced complexity of the underlying cores. Traditionally, GPUs implement many-core architectures and are used for rendering various graphics scenes. Recently there have been improvements and the graphics cards have been adapted such that they can run even general purpose code efficiently [2]. This involved a number of architectural changes, such as having write caches, instead of read-only. GPUs have the following main structure: they contain Streaming Multiprocessors (SMs), which in turn contain a number of Streaming Processors (SPs). As a result, a graphics card can end up being able to process thousands of pieces of data simultaneously. The challenge that comes with this is the ability of the developers to exploit this massive parallelism and not waste resources. Another feature specific to GPUs is that each SM can only execute one instruction at a time, therefore all its SPs execute the same instruction or none at all. This offers another challenge for the programmers, especially in the cases of conditionals, when some elements evaluate to one branch, whilst the rest to the other one. More

information about GPU architectures can be found on CUDA Zone [9] or AMD's website [10].

2.2 Languages

2.2.1 CUDA

CUDA (Compute Unified Device Architecture) [11] [9] is developed by NVIDIA to be a parallel computing platform and an associated programming model. It can be used in conjuncture with C/C++ by using "C for CUDA". It allows the developers to manipulate lower level features of the hardware, such as memory transfers between global and device memory [12]. When well used, these lead to a significant increase in performance, as specifics of devices they run on can be used to optimize the application. Currently, only NVIDIA devices have CUDA support. Alternatives for CUDA are DirectCompute [13] and OpenCL, which run on multiple platforms, including AMD graphics cards.

2.2.2 OpenCL

OpenCL is an open, royalty-free standard for a low-level computing platform started by Apple which is currently implemented by IBM, Intel, AMD and Nvidia. It offers an alternative to platform-specific frameworks such as CUDA, given that it is supported by both CPUs and GPUs. OpenCL follows a similar programming model to CUDA and offers C/C++ extensions [14]. A reason why OpenCL is still not the primary language used is due to the fact that the standard is still relatively new and there are still problems with the existing implementations.

2.3 OP2

OP2 is a library and framework that provides a high-level abstraction for the parallel processing of unstructured mesh applications. It offers a set of instructions that can be used to declare the mesh, dependencies between elements and data related to them. It also provides specific `for` loops, which take in data they work on and user supplied kernels that specify operations to be performed on the data [15] [16]. One very important property of the `for` loops is that the same results will be achieved, no matter in what order the provided data is processed. Starting from this premise, the actual optimization choices and target architectures are left as a choice to the library,

which provides an execution plan for each loop. In Listing 2.1 you can see a sample of the Airfoil code which uses OP2. The full version of the OP2 Airfoil C++ code, including the user-defined kernels, can be found in Appendix A.1.

Listing 2.1: Code sample from Airfoil illustrating the OP2 library statements.

```
int main(int argc, char **argv) {
  // OP initialisation
  op_init(argc,argv,2);

  // declare sets, pointers, datasets and global constants
  op_set nodes  = op_decl_set(nnode,  "nodes");
  op_set edges  = op_decl_set(nedge,  "edges");
  op_set bedges = op_decl_set(nbedge, "bedges");
  op_set cells  = op_decl_set(ncell,  "cells");

  op_map pedge   = op_decl_map(edges, nodes,2,edge,  "pedge");
  op_map pecell  = op_decl_map(edges, cells,2,ecell, "pecell");
  op_map pbedge  = op_decl_map(bedges,nodes,2,bedge, "pbedge");
  op_map pbecell = op_decl_map(bedges,cells,1,becell,"pbecell");
  op_map pcell   = op_decl_map(cells, nodes,4,cell,  "pcell");

  op_dat p_bound = op_decl_dat(bedges,1,"int"  ,bound,"p_bound");
  op_dat p_x     = op_decl_dat(nodes ,2,"float",x    ,"p_x");
  op_dat p_q     = op_decl_dat(cells ,4,"float",q    ,"p_q");
  op_dat p_qold  = op_decl_dat(cells ,4,"float",qold ,"p_qold");
  op_dat p_adt   = op_decl_dat(cells ,1,"float",adt  ,"p_adt");
  op_dat p_res   = op_decl_dat(cells ,4,"float",res  ,"p_res");

  op_decl_const(1,"float",&gam  );
  ...

  // main time-marching loop
  niter = 1000;

  for(int iter=1; iter<=niter; iter++) {

    // direct loop
    op_par_loop(save_soln,"save_soln", cells,
                op_arg_dat(p_q,   -1,OP_ID, 4,"float",OP_READ ),
                op_arg_dat(p_qold,-1,OP_ID, 4,"float",OP_WRITE));

      for(int k=0; k<2; k++) {

        op_par_loop(adt_calc,"adt_calc",cells,
                    op_arg_dat(p_x,   0,pcell, 2,"float",OP_READ ),
                    op_arg_dat(p_x,   1,pcell, 2,"float",OP_READ ),
                    op_arg_dat(p_x,   2,pcell, 2,"float",OP_READ ),
                    op_arg_dat(p_x,   3,pcell, 2,"float",OP_READ ),
                    op_arg_dat(p_q,  -1,OP_ID, 4,"float",OP_READ ),
```

```
45                      op_arg_dat(p_adt,-1,OP_ID, 1,"float",OP_WRITE));
46
47          // indirect loop
48          op_par_loop(res_calc,"res_calc",edges,
49                      op_arg_dat(p_x,    0,pedge, 2,"float",OP_READ),
50                      op_arg_dat(p_x,    1,pedge, 2,"float",OP_READ),
51                      op_arg_dat(p_q,    0,pecell,4,"float",OP_READ),
52                      op_arg_dat(p_q,    1,pecell,4,"float",OP_READ),
53                      op_arg_dat(p_adt,  0,pecell,1,"float",OP_READ),
54                      op_arg_dat(p_adt,  1,pecell,1,"float",OP_READ),
55                      op_arg_dat(p_res,  0,pecell,4,"float",OP_INC ),
56                      op_arg_dat(p_res,  1,pecell,4,"float",OP_INC ));
57        ...
58      }
59      ...
60    }
61    ...
62  }
```

The front-end of the library is currently in C, C++ or Fortran, and there are currently back-end implementation for CUDA and OpenMP [17]. OpenCL and AVX/SSE implementations under C++ are currently in development.

One of the main purposes of the library is to abstract away detailed implementation choices from the user and allow them to program by using a more familiar and easily understandable interface. This also gives a lot of freedom to the OP2 developers, as to the choice of optimizations and the target architectures. As a comparison, languages such as C, C++ offer little exposure to the underlying architecture; therefore platform-specific optimizations cannot be created easily. For most pieces of software available, that is sufficient. However, for a subset such as scientific or engineering simulators, that does not render the desired performance. The next step is to use a language such as OpenCL, which exposes a lot more functionality of the underlying architecture. This is what we chose to use with OP2, as it provides the most flexibility in the choice of optimizations and target platforms. A further possibility would be to use CUDA or, for Intel CPUs, AVX/SSE intrinsics, however, this would add far too much complexity to the project and would prove a distraction from our goals. Furthermore, there already exist CUDA back-end implementations for OP2.

Similar to OP2, there exists a domain-specific language developed at Stanford, called Liszt [18]. It is a Scala [19] based language used for writing mesh-based applications used to solve partial differential equations. Given its more targeted audience, Liszt offers a wider range of features, such as already-implemented mesh operations (example: a Jacobi iteration), in addition to the instructions to declare the meshes. Just as OP2, it offers parallel `for`-

comprehension loops that do not contain any loop-carried dependencies. Its advantage is that by using a domain-specific constructs, it can better choose the implementation for each of them. In OP2, we compensate for their lack by using machine learning algorithms that determine the best set of optimization choices for a given problem.

More specifically, for each problem there exists an optimal set of optimizations for a particular architecture. With the aid of the machine learning algorithm, we carefully choose those, as when applying multiple levels of optimizations, some might interfere with the efficiency or application of subsequent ones. Therefore, we need to ensure that the chain of optimizations is optimal for the chosen target device.

In order to achieve the aim of having optimal optimizations, we need to maximize data reuse and parallelism. In both OP2 and Liszt, the latter is obtained by making use of partitions and colouring. The data is partitioned such that if two edges want to update the same node, only the owner of the node data performs the update. Redundancy appears when a node has multiple owners. Graph colouring is used to ensure that no two edges update the same node. Maximum data reuse can be obtained by reordering the iterations to make sure that two consecutively computed partitions make use of the same nodes and edges.

2.3.1 Direct and Indirect loops

As the main performance issues to be tackled are generated by the `for` loops, we will describe them in more detail. Each iterates over a given set (example from Listing 2.1: `edges`) and accesses related data through mapping arrays (example from Listing 2.1: `pedge`), which it then processes and writes back to some data array. If there is no level of indirection (i.e. accessing data through no mapping array), then the loop is direct. If there a level of indirection, then it is an indirect loop. OP2 currently limits the number of indirection levels to one. This is due to the fact that each level of indirection generates a loss of parallelism by increasing the data dependency.

Direct loops only access data directly related to the elements being processed. For example, if we iterate over `nodes`, then we write to the node data `p_x`, and we not use explicitly a map, as this is a direct access. Therefore, we have only used one level of indirection. An example of this can be seen in Listing 2.1, by looking at the `save_soln` loop.

If, however, we want to loop over `cells` and write over node data in `p_x`, we would then have to use one mapping to `pcell`, therefore we would have one level of indirection and an indirect loop. Furthermore, given that two different cells can have the same nodes, the data dependency earlier mentioned arises

and then various optimization techniques can be used to minimize its effects. An example of an indirect loop is `res_calc` in Listing 2.1.

2.4 Optimization Techniques

2.4.1 Graph colouring

Graph colouring is a parallelization technique that labels with the same colour all elements of a dataset that can be processed concurrently. In the case of an unstructured mesh that has as a base element a triangle, any two adjacent triangles will have different colours, as a data dependency is created due to the shared edges. By applying this, we allow for a parallel execution of same colour elements, followed by a synchronization step. Figure 2.1 shows an example of how a graph is coloured.

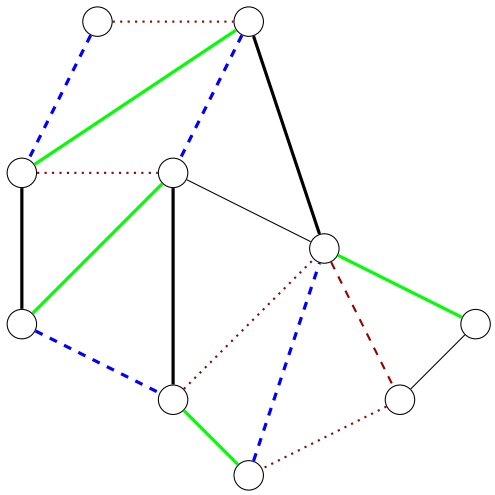


Figure 2.1: Example of graph colouring on a triangle-based unstructured mesh.

2.4.2 Iteration reordering

In order to reuse as much of the data in local memory as possible, one way is to reorder the iterations that apply to it. Within the Airfoil example, we have 5 loops within each time step, and each iterates over the same data. In this case, we can merge and rearrange the loops to compute all kernels related to a particular subset of data at a time. If we choose to not merge the loops, but keep them as they are, we can still reorder the iterations by

making sure we that if now we compute the first set of data, say coloured red, then immediate set will be blue, where all the elements in the second set are adjacent to the first set. This will ensure that the underlying data that is used by the triangles is reused, as it is the same.

2.4.2.1 Polyhedral Models

A polyhedral model is a mathematical representation used for optimizations regarding nested loops. The basis of the model relies on it treating each nested loop iteration as a grid point inside a polyhedron, and then performs transformations (such as affine or tiling) on a mesh which contains the polyhedra. The result is a mesh representing an optimized and equivalent iteration space of the nested loops.

Such a model is of great value in computer optimizations as it offers an abstract model for reasoning with nested loop transformations that is easily used with a great number of changes. In practice, attempting to do the same without such a model can prove impossible, due to the complexity of the transformations when they are not abstracted away.

Sparse computations using polyhedral frameworks are slow, as the generated models do not take into account the properties of the data. As a result, a whole set of sparse computations, such as finite element analysis, are inefficient when using traditional models. This also applies in our case, as each element we iterate over is connected to very few others, thus generating sparse connectivity matrices. A solution to this was developed by Michelle M. Strout et al. from Colorado State University who introduced the Sparse Polyhedral Framework (SPF) [20] [21].

One of the main challenges in traditional approaches is the inability to properly reorder data and computation at compile-time in such applications. Another issue is that polyhedral models use Inspector/Executor strategies [22] [23], however, few have been automated. SPF overcomes these issues by choosing run-time reordering transformations and by adding uninterpreted functions to the polyhedral framework. The end result is inspector/executor code.

Each statement in SPF has the following representation: an iteration space, a scheduling function and access functions for the indirect references to the data array. This specification is currently being generated with IEGenCC [24]. From those, SPF generates a data space specification and the uninterpreted functions. From these, a transformation specification is constructed represented as tuple relations of the data and iteration reordering. IEGen [24] generates the inspector/executor code from this specification, by using a full sparse tiling strategy [25] [26].

Main features of SPF and IEGen are that they traverse the access relations and data dependencies, then generate optimization changes for them, iteratively. This results in a refined model, which is then transformed into code in the following way: Cloog [27] is used for the outer loops, whilst the framework deals with the sparsity in inner loops and access relations.

The overall process for generating inspector/executor code for sparse polyhedral models is:

- identify the indirect nested loops with IEGenCC
- enable the specification of computations and run-time reordering transformations with SPF
- sparse tilings are computed at run-time with transformation writers
- generate the inspector/executor code with IEGen

The full sparse tiling algorithm splits sparse matrices into tiles, which can be further parallelised by using graph colouring, and then their size can be adjusted in order to improve parallelism. One of the main advantages of this algorithm is that it turns data reuse into data locality, thus improving the overall execution time of an application.

2.4.3 Loop fusion

Loop fusion is an optimization which involves combining two or more loops into one, as seen in Listing 2.2.

Listing 2.2: Original versions of the save_soln and adt_calc loops, and then a fused version of the two. Loop fusion is possible here as we can see that they use nearly all of the same parameters, and thus it is advantageous to fuse the loops.

```
1
2  // Original code of op_par_loop save_soln and adt_calc.
3
4        op_par_loop(save_soln,"save_soln", cells,
5            op_arg_dat(p_q,   -1,OP_ID, 4,"float",OP_READ ),
6            op_arg_dat(p_qold,-1,OP_ID, 4,"float",OP_WRITE));
7
8        op_par_loop(adt_calc,"adt_calc",cells,
9            op_arg_dat(p_x,   0,pcell, 2,"float",OP_READ ),
10           op_arg_dat(p_x,   1,pcell, 2,"float",OP_READ ),
11           op_arg_dat(p_x,   2,pcell, 2,"float",OP_READ ),
12           op_arg_dat(p_x,   3,pcell, 2,"float",OP_READ ),
```

```
13            op_arg_dat(p_q,   -1,OP_ID, 4,"float",OP_READ ),
14            op_arg_dat(p_adt,-1,OP_ID, 1,"float",OP_WRITE));
15
16  // A fused version of the above loops.
17
18        op_par_loop(save_soln_adt_calc_fused,
19            "fused_save_soln_adt_calc", cells,
20            op_arg_dat(p_x,    0,pcell, 2,"float",OP_READ ),
21            op_arg_dat(p_x,    1,pcell, 2,"float",OP_READ ),
22            op_arg_dat(p_x,    2,pcell, 2,"float",OP_READ ),
23            op_arg_dat(p_x,    3,pcell, 2,"float",OP_READ ),
24            op_arg_dat(p_q,   -1,OP_ID, 4,"float",OP_READ ),
25            op_arg_dat(p_qold,-1,OP_ID, 4,"float",OP_WRITE),
26            op_arg_dat(p_adt,-1,OP_ID, 1,"float",OP_WRITE));
```

This can significantly increase performance, given that instead of copying an array of data, portion by portion and processing it, then copying it again in memory and doing another set of operations, it is beneficial to perform both operations for each portion, as it is in memory. Figure 2.2 contains an example.

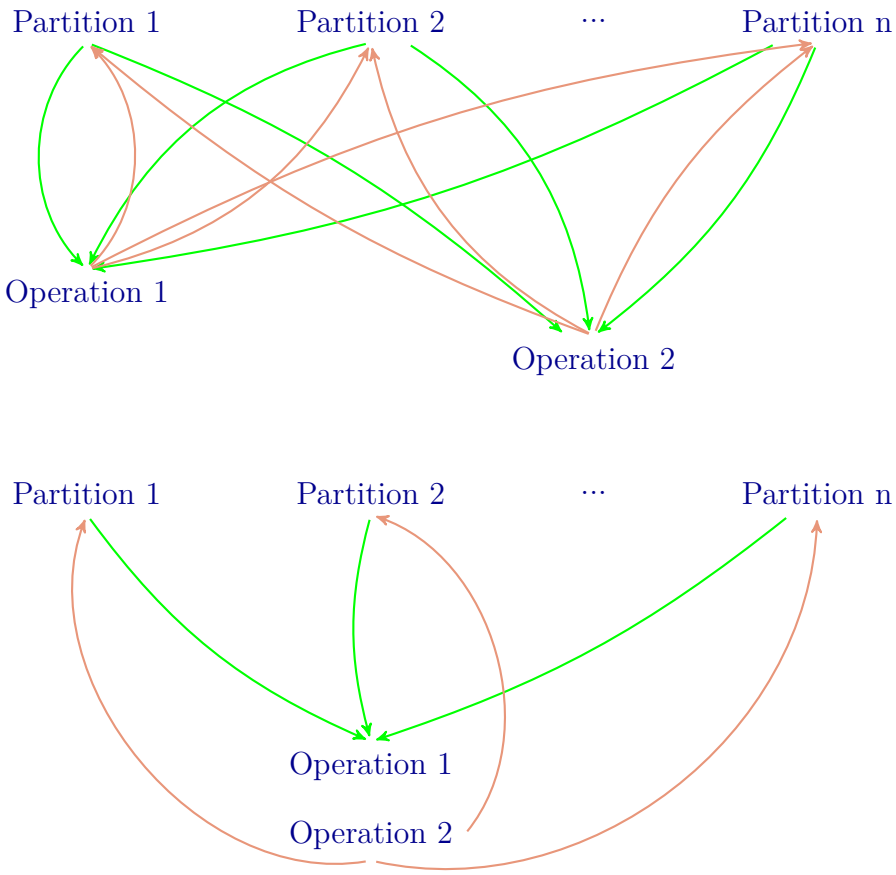


Figure 2.2: The top graph shows the memory transfers between the partitions, for each operation. In this case, each operation is equivalent to a computational kernel. We observe that the same data is resent to device for each operation, then copied back, in order (operation 1 first, then for operation 2). We can tell that this has double overhead, compared to the fused version which is represented in the second graph. In the bottom graph we observe that for each partition, we only copy it once, perform all operations and then move it back.

2.4.4 Loop unrolling

Loop unrolling is an optimization technique that involves extracting a loop's body outside of the loop n-times, where n represents the number of iterations. The order of operations is preserved, and thus, the code's output. In Listing 2.3 we can see an example of loop unrolling.

Listing 2.3: A generic example of loop unrolling. Here we see that it involves copying the body of the loop n-times, where n is the loop's number of iterations. This optimization helps reduce the number of loop control arithmentic, thus reducing branch penalties.

```
// Original loop

for (int i = 0; i < n; ++i) {
  Statement1;
  Statement2;
  ..
  StatementM;
}

// Unrolled loop - we have n groups of Statements 1-M

Statement1;
Statement2;
...
StatementM;

Statement1;
Statement2;
...
StatementM;

...
```

This can be used in conjuncture with other optimizations, such as loop fusion. In OP2, the number of possible, efficient loop fusions is 1, until we perform a loop unrolling, which unlocks another loop fusion optimization. Listing 2.4 shows this.

Listing 2.4: Original versions of the save_soln and adt_calc loops, and then a fused version of the two. The loop fusion is possible here as we can see that they use nearly all of the same parameters, and thus it is advantageous to fuse the loops.

```

// Original version of save_soln and adt_calc
n = 1000;

```

```
for (int i = 0; i < n; ++i) {
  op_par_loop("save_soln");
  for (int j = 0; j < 2; ++j) {
    op_par_loop("adt_calc");
    op_par_loop("res_calc");
    op_par_loop("bres_calc");
    op_par_loop("update");
  }
}

// Inner loop is unrolled and we fused save_soln
// and adt_calc

n = 1000;

for (int i = 0; i < n; ++i) {
  op_par_loop("fused_save_soln_adt_calc");
  op_par_loop("res_calc");
  op_par_loop("bres_calc");
  op_par_loop("update");
  op_par_loop("adt_calc");
  op_par_loop("res_calc");
  op_par_loop("bres_calc");
  op_par_loop("update");
}
```

2.4.5 Code-block reordering

Code-block reordering is a technique that involves re-arranging the code's basic blocks in order to improve locality of reference and to reduce branching. This optimization can be successfully used with others, such as loop unrolling and loop fusion. By reordering the blocks, we can create loop fusion opportunities, as seen in Listing 2.5.

Listing 2.5: In this code example we show a code-block reordered and simplified version of the Airfoil. The next part of the code example contains a fully optimized Airfoil, which adds loop unrolling and fusion. This example clearly shows the previously unexistent fusion possiiblities without having to unroll the outside loop.

```

// For this example we use a reordered, simplified version of the Airfoil.
// This is a code-block re-ordered version, without fusion.
n = 1000;
op_par_loop("save_soln");

for (int i = 0; i < n-1; ++i) {
```

```
8     for (int j = 0; j < 2; ++j) {
9       op_par_loop("adt_calc");
10      op_par_loop("res_calc");
11      op_par_loop("bres_calc");
12      op_par_loop("update");
13    }
14    op_par_loop("save_soln");
15  }
16
17  for (int j = 0; j < 2; ++j) {
18      op_par_loop("adt_calc");
19      op_par_loop("res_calc");
20      op_par_loop("bres_calc");
21      op_par_loop("update");
22  }
23
24  // Now we have added loop fusion and loop unrolling.
25
26  op_par_loop("save_soln")
27
28  for (int i = 0; i < n-1; ++i) {
29      op_par_loop("adt_calc");
30      op_par_loop("res_calc");
31      op_par_loop("bres_calc");
32      op_par_loop("fused_update_adt_calc");
33      op_par_loop("res_calc");
34      op_par_loop("bres_calc");
35      op_par_loop("fused_update_save_soln");
36  }
37
38  for (int j = 0; j < 2; ++j) {
39      op_par_loop("adt_calc");
40      op_par_loop("res_calc");
41      op_par_loop("bres_calc");
42      op_par_loop("update");
43  }
```

2.4.6 Spatial and Temporal Locality

Spatial locality is based on the premise that, if a particular memory location is accessed at some time, then nearby locations are very likely to be accessed in the immediate future. For this situation, it is good to make sure that all these locations are found in the local memory to not waste time retrieving them from memory. For an unstructured mesh with a triangle base, we can explore spatial locality by making sure that, for example, all edges of a triangle are in memory. If one of them is being accessed at a particular time, it is very

likely that all of them will.

Temporal locality is a special type of spatial locality, in which a prospective location is the same as the actual one, and that relies on a particular memory location being accessed multiple times in the near future. In the case of the mesh, we can see this as executing a kernel over two sets of adjacent triangles that use the same node and edge information.

2.4.7 Memory organization – Array of Structures vs. Structure of Arrays

When dealing with structured data (example: data representing triangles), there are two ways of representing it: Array of Structures and Structure of Arrays.

For unstructured meshes with a base element of a triangle, we represent coordinates of nodes and edges for each of them. This can be seen in Figure 2.3.

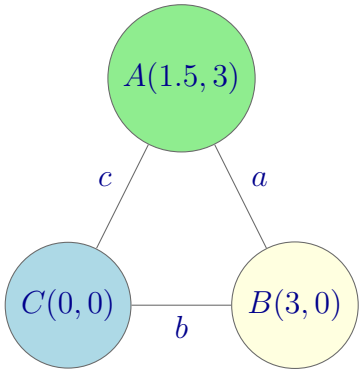


Figure 2.3: Example of an unstructured mesh triangle with node and edge data.

In the case of array of structures, each entry in the array represents a triangle structure. It contains a list of nodes and a list of edges. Examples can be seen in Listing 2.6 and Figure 2.4.

Listing 2.6: Array of Structures code example

```
typedef struct triangle {
  int nodeA[2];
  int nodeB[2];
  int nodeC[2];
  int edgeA;
  int edgeB;
  int edgeC;
}
```

```
9
10 int main(int* argc, int** argv[]) {
11   ...
12   triangle meshData[n];
13   ...
14 }
```

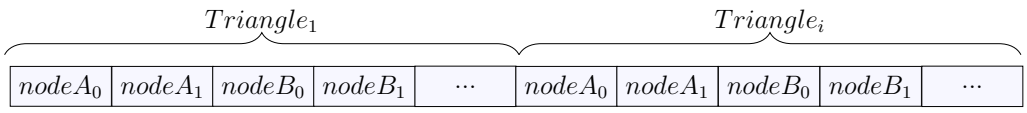


Figure 2.4: One-dimensional Array of Structures in-memory representation example.

If the organisation is structure of arrays, then we would have a structure, with the following elements: one array for all top nodes, one for all right nodes and one for all left nodes. Analogous arrays exist, as part of the structure, for the edges. Examples can be seen in Listing 2.7 and Figure 2.5.

Listing 2.7: Structure of Arrays code example

```
1  typedef struct triangle {
2    int nodeA[2,n];
3    int nodeB[2,n];
4    int nodeC[2,n];
5    int edgeA[n];
6    int edgeB[n];
7    int edgeC[n];
8  }
9
10 int main(int* argc, int** argv[]) {
11   ...
12   triangle meshData;
13   ...
```

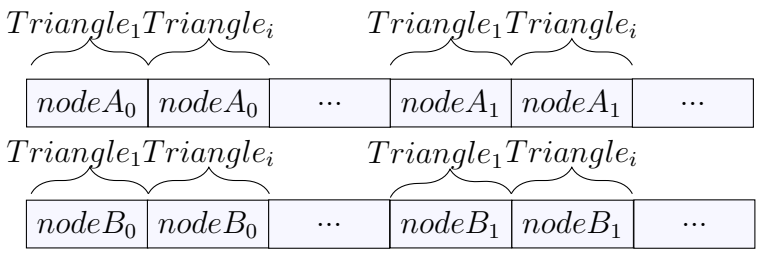


Figure 2.5: One-dimensional Structure of Arrays in-memory representation example.

Each representation traditionally offers advantages and disadvantages on each type of architecture (CPU or GPU). With modern developments, it is not fully clear which option is best in which case, therefore, we are exploring both in this project. Traditionally, due to the significant parallelism offered by GPUs, a structure of arrays representation was chosen, whilst for CPUs, array of structures is more commonly met.

2.5 Machine Learning Techniques

2.5.1 Overview

Machine learning is a branch of artificial intelligence that focuses on designing algorithms that describe evolving behaviours based on empirical data. In order for a learning problem to be well-posed, it needs to satisfy three main requirements: it needs a measure of success, experience gained through training examples and the ability to better its results with experience. A widely accepted definition by Tom M. Mitchell [28]:

> *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

Every machine learning system contains the following main steps: it takes a new problem (by creating it or by having it as a given), it uses the current knowledge to come up with a solution, then it analyses how good the result is and finally, it creates a hypothesis based on the example. By using this procedure, the system keeps adapting itself to based on each experience. Within this, there exist variations based on whether the system is eager or lazy to learn. The main difference is that lazy systems just store data and only generalize beyond it until there is an actual request, whilst eager ones construct general, explicit descriptions of their target function (problem to solve), based on already provided training examples. This difference makes lazy learning to be very suitable for problem domains that are either complex or incomplete, where their target function can be represented by a collection of less complex smaller approximations.

2.5.2 Case-based Reasoning Systems

Case-based reasoning is a lazy machine learning technique that relies on finding solutions for new similar problems based on a set of previously solved problems.

It is based on work by Roger Schank [29] [30], which was inspired by findings in cognitive sciences on human reasoning and memory organization. All similar concepts or experiences of a human are organized in memory packets. When a person experiences something new and there already exists a memory packet that has successfully solved a similar problem, then the previous experience is recollected and the same steps are followed to reach a solution. Therefore, case based reasoning systems reapply previously successful solution schemes to find an answer for a new problem, instead of applying a general set of rules.

Schank's memory-based reasoning model contains the following steps: there is a set of observed cases stored in memory organization packets, from which the memory of experiences is derived. Then if the new experience matches an existing one, then that one will be retrieved, otherwise, we use similarities to come up with a solution. This memory-based model follows an automatic, online learning technique, which implies that we keep learning with every new case, as opposed to typical eager systems, where we cease to learn when the training is complete.

Algorithmically, we will have the following steps, also seen in Figure 2.6:

1. search through known cases for a similar instance
2. retrieve these instances
3. adapt the solutions to the current one
4. add the new solved case to the known experiences

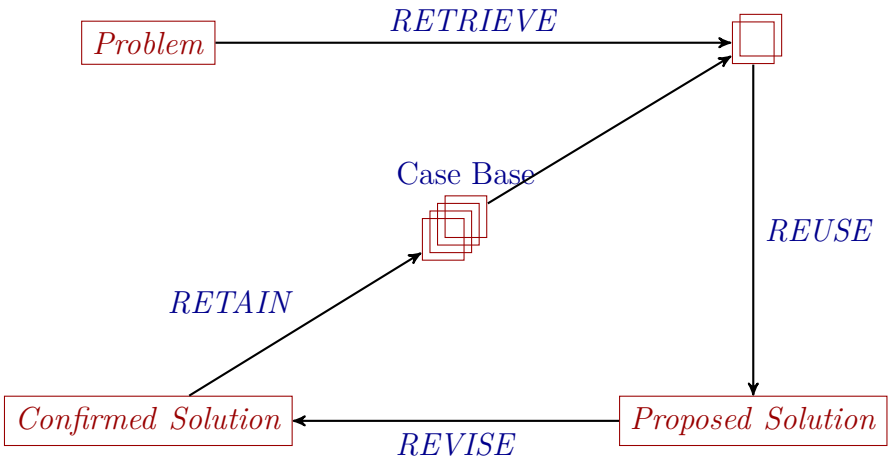


Figure 2.6: A representation of the steps employed by a case-based reasoning system [28], from the moment a problem arrives, to the retrieval and choice of a solution, and then saving the current experience in the database of cases.

2.5.2.1 k-Nearest Neighbour Learning

For the retrieval step of the most similar cases, we can use the k-Nearest Neighbour learning algorithm [31]. The list of possible nearest neighbours is the one of previously observed instances x_i, $i = [1..n]$, with n being the number of instances. Given query x_q, we can select the most similar cases by using an Euclidean distance $d(x_i, x_q)$. As a refinement of this algorithm, we can assign weights w_i to each neighbour x_i of the query instance x_q based on the distance $d(x_i, x_q)$, such that $(d(x_i, x_q) \downarrow \longleftrightarrow w_i \uparrow)$.

With this refinement, we get: given a target function $V : X \to C$ and a set of n observed instances (x_i, c_j), where $x_i \in X$, $i = [1..n]$, $c_j \in C$, $j = [1..m]$, $V(x_i) = c_j$, distance weighted k-NN algorithm will decide the class c_l, $l = [1..m]$, of the query instance x_q based on its k nearest neighbours x_r, $r = [1..k]$, in the following way:

$$V(x_q) \leftarrow c_l \in C \longleftrightarrow \forall (j \neq l) \sum_r w_r * E(c_l, V(x_r)) > \sum_r w_r * E(c_j, V(x_r))$$

$$\text{where } E(a, b) = 1 \text{ if } a = b, \text{ else } 0, \text{ and } w_r = 1/(d(x_r, x_q))^2$$

One of the main advantages of the refined algorithm is that it is more robust to noisy training data, as it calculates $V(x_q)$ based on weighted $V(x_r)$ values of all its k nearest neighbours x_r, thus eliminating the effects of noisy data. However, as the algorithm calculates the distance between instances

based on all the attributes, including irrelevant ones, we can end up with the wrong classification. As a result, we can just weight each attribute differently, based on its importance for the particular class.

2.6 Summary

In this chapter we detailed on the main tools and techniques that we use throughout this project, to offer a better insight into the aims and complexity of the work. We detailed on the OP2 framework in which all the work is integrated, a set of possible optimization techniques, languages and the machine learning algorithm we use for the purpose of this project.

Chapter 3

Related work

The idea of having an optimizing compiler that can learn from its past experiences and choose the best speed-ups is not a new idea. With the recent developments in microarchitectures, the requirement for such frameworks and libraries has increased significantly, as the best options of optimizations and architectures are no longer as clear.

As mentioned in the OP2 section, Stanford's Liszt framework is a very similar language to OP2, that employs the same `for`-comprehension loops and domain-specific language for declaring the necessary data structures (nodes, edges, maps etc.). Liszt offers further statements that concern the actual processing of the data, unlike OP2, which relies on user-supplied kernels that describe the operations that need to be performed on the data. OP2's approach gives the user more flexibility, as they can define operations not included by default in Liszt. Nevertheless, both frameworks rely on the same ideas of running unstructured mesh programs across heterogeneous platforms optimally and abstracting any low-level and platform-specific implementation details from the user. This approach ensures maximum reuse of the user's code.

With regards to choosing which optimizations are best for a particular architecture by making use of learning algorithms, researchers from University of Edinburgh have presented a portable optimising compiler [32] that can automatically adapt to underlying microarchitectural changes. Currently, it is only targeted at multiple generations of a microprocessor, however, it represents the first step to a universal compiler that can target any platform without requiring extensive tuning. Its machine learning algorithm is based upon building a model that provides a mapping from a given set of program/microarchitecture properties to a set of good optimisations passes. They learn the mapping from the properties to a probability distribution over good optimisation passes. Afterwards, they sample at the mode of the

distribution to obtain a prediction on a new program and on a new microarchitecture. They obtain the predicted set of optimizations by finding the set that gives the greatest probability of being a good optimisation.

For the CUDA-back-end of OP2, there exists an auto-tuner called Flamingo [33], that chooses the optimum optimization for each kernel in turn. However, the tuner, written in Python, uses approach close to a brute force one, which severely restricts the number of available applications. Furthermore, it relies on all of its input parameters, such as size of partition, number of threads in a block etc. of being independent. This also limits the number of parameters that can be used. The application builds a tree of variable dependencies, where siblings in the tree are considered independent. The algorithm is exponential with respect to the depth of the tree.

Flamingo and the optimising compiler offer a good foundation and a promising premise for this project. They show that there is not just room for improvement, but a strong possibility that a machine learning algorithm that chooses between different optimizations, across multiple platforms, used in conjunction with OP2, can achieve a speed-up over traditional hand-tailored approaches. Furthermore, they both show that there is a correlation between a program's various parameters and its performance, when run on a particular architecture.

3.1 Research goals

Within this project, we explore a different perspective that combines ideas from Flamingo and Edinburgh's optimising compiler. Here we use machine learning algorithms, instead of a brute force approach, to determine the best set of optimizations for a given program on a particular architecture, from a number of speed-ups and target platforms. This allows us to generalize over the optimising compiler and improve on the search space compared to Flamingo. Given OP2's structure that relies on user supplied kernels for the computation, we obtain maximum freedom in choosing the optimization choices such as loop fusion and runtime parameter tweaking. We also use a different approach, compared to Edinburgh's, with respect to deducing optimal sets of optimizations from a given program and architecture. They use detailed low level architectural information, which we do not include, and derive the optimizations statistically. We use a weighting system applied to a vector of properties that gives us the desired runtime parameter and optimization suggestions.

As a result, we aim to determine whether there is a generalized connection between various program and architecture parameters from which we can

learn and that can lead to better overall optimization and platform choices.

3.2 Summary

In this chapter we presented related work and offered details on how this project builds up on it, thus offering a novel proof of concept.

Chapter 4

Tools: OP2 OpenCL runtime and compiler

In order to automatically generate Airfoil variations which we require for testing the machine learning algorithms presented in Section 5.5, we require a fully working up-to-date OpenCL runtime and OP2 to OpenCL compiler.

In this chapter we detail on the work that had to be done on existing tools that we used for the completion of the project. All code and experiments mentioned in this chapter have been run on the following hardware and software configurations: AMD , NVIDIA and Intel OpenCL 1.1, and NVIDIA Tesla M2050 GPU, Intel Xeon X5650 and Intel Core i7 2620M.

4.1 OpenCL Airfoil and OP2

We have decided to use OpenCL as the back-end for the compiler because the generated code runs on both CPUs and GPUs without further modifications. This meant that we needed both an OpenCL runtime and for the OP2 compiler to generate working OpenCL code. The advantage of this back-end was that tools and implementations already existed for both the runtime and the compiler. Now we will detail on the code's structure and on the work that had to be done in order to ensure these results.

4.1.1 OP2 OpenCL Airfoil structure

The OP2 OpenCL Airfoil is composed of:

- `airfoil.cpp` – this file contains all the OP2 code and the `op_par_loop` calls of the 5 Airfoil loops.
- Host file(s) – Host code can be found in one or more files (one containing all host codes, or one file for each loop host code). In the OpenCL implementation, the host code is responsible for calling the respective kernels with appropriately set parameters. The host code gets executed on the processor.
- Kernel file – contains all the computational kernels, including the user supplied ones as `inline` functions. These are the kernels that actually execute on the device.

4.1.2 Runtime

For the runtime we have used an existing OpenCL implementation of Airfoil written by a past Imperial College London MSc student. The advantages of this implementation are that it already contained all the extra runtime code needed to ensure the proper initialization of the OpenCL code, appropriate host and kernel methods, all in the context of the OP2 runtime. The disadvantages of this codebase were that it initially gave wrong results when the code was run on CPUs and that it was tightly integrated with an older version of the OP2 runtime.

The first step was to get the code to work on a CPU, as part of the project's goals is to decide on the optimizations based on a given architecture. When initially run, the given code gave `NaN` results. On GPUs, no issues with the results were observed. For each loop, we considered arrays that were written to as our result arrays. We saw that the results went wrong after the `save_soln` loop shown in Listing 4.1.

Listing 4.1: Initial save_soln code sample

```
1
2 for (int n = get_global_id(0); n < set_size; n+=get_global_size(0)) {
3   int offset = n - tid;
4   int nelems = MIN(OP_WARPSIZE, set_size-offset);
5   for (int m = 0; m < 4; m++) {
6     arg_s[tid+ m*nelems] = arg0[offset*4 + tid + m*nelems];
7   }
8
9   for (int m = 0; m < 4; m++) {
```

```
10      arg0_l[m] = arg_s[tid*4   + m];
11    }
12
13    save_soln (arg0_l, arg1_l);
14  }
```

After careful inspection of the code, we observed two issues: memory accesses to outdated memory locations. This was likely caused by the fact that on a CPU threads do not run in lockstep. In order to access the memory locations written to by other threads, we first need to synchronize all threads to make sure all write operations have completed before other threads read those memory locations. This can be achieved by adding barriers and we have used `barrier(CLK_LOCAL_MEM_FENCE)`, as seen in 4.2.

Listing 4.2: save_soln with barriers code sample

```
1
2  for (int n = get_global_id(0); n < set_size; n+=get_global_size(0)) {
3    int offset = n - tid;
4    int nelems = MIN(OP_WARPSIZE, set_size-offset);
5    for (int m = 0; m < 4; m++) {
6      arg_s[tid+ m*nelems] = arg0[offset*4 + tid + m*nelems];
7    }
8
9    barrier (CLK_LOCAL_MEM_FENCE);
10
11   for (int m = 0; m < 4; m++) {
12     arg0_l[m] = arg_s[tid*4   + m];
13   }
14
15   barrier (CLK_LOCAL_MEM_FENCE);
16
17   save_soln (arg0_l, arg1_l);
18 }
```

At this stage we observed correct values, however, the last two values in the result array were not written to. This likely meant that the kernel was suffering from thread divergence. Thread divergence occurs when in a block, some threads evaluate a conditional expression to true, some to false. In this context, this came in the shape of `for` loop that iterated through all elements of the array. The code assumed that there are 4 threads in a block (through `OP_WARPSIZE`), however, the number of elements in the array was not a multiple of 4. As a result, when we had to update the last two elements, two of the threads entered the `for` loop, whilst two did not. Furthermore, the code had barriers which have the property that all threads in a block must execute the exact same barriers, otherwise they can block. As a result, the code that was supposed to execute the update of the result array elements

was not reached, as the threads would block, waiting at the barrier for the two threads that did not actually even enter the `for` loop. A solution for this problem is to let all threads enter the loop and add appropriate `if` statements around computational blocks, whilst the barriers stay on the outside of those blocks. This ensures that all threads execute the same barriers and the thread block cannot block. This can be seen in Listing 4.3.

Listing 4.3: save_soln with barriers and thread divergance safety code sample

```
1
2  for (int n = get_global_id(0); n < set_size + set_size % OP_WARPSIZE;
3       n+=get_global_size(0)) {
4    int offset = n - tid;
5    int nelems = MIN(OP_WARPSIZE, set_size-offset);
6    for (int m = 0; m < 4 && n < set_size; m++) {
7      arg_s[tid+ m*nelems] = arg0[offset*4 + tid + m*nelems];
8    }
9
10   barrier (CLK_LOCAL_MEM_FENCE);
11
12   for (int m = 0; m < 4 && n < set_size; m++) {
13     arg0_l[m] = arg_s[tid*4  + m];
14   }
15
16   barrier (CLK_LOCAL_MEM_FENCE);
17
18   if (n < set_size) {
19     save_soln (arg0_l, arg1_l);
20   }
21 }
```

At this stage in the debugging process, the code should have worked, however, despite not having any `NaN` results, the output values were incorrect. After careful further examination of the code, we deduced that the wrong results were due to `OP_WARPSIZE`. This variable shows how many threads run in lockstep. As we were running the code on a CPU, and we observed that at any time, it was running on a maximum number of cores, we assumed that we can simulate lockstep by using barriers and capturing thread divergence cases. We also assumed that OpenCL's workgroups were equal to the maximum number of cores. If this were true, the above methods should have fixed the problem. However, this assumption was wrong and the workgroups only contained 1 work-item, thus only 1 thread. This meant that the hard-coded value of `OP_WARPSIZE = 4` was wrong, and was causing the wrong results. In Listing 4.4 we give a sample of code that is broken if `OP_WARPSIZE` and the workgroup size are not the same. In Figure 4.1 we provide an explanation as to why this is the case.

Listing 4.4: adt_calc code sample that breaks when `OP_WARPSIZE` and the workgroup size are not the same.

```
1
2  // This is a code sample from adt_calc
3  int    tid = get_local_id(0)%OP_WARPSIZE;
4
5    __local float *arg_s =  shared+ offset_s *(get_local_id(0)/OP_WARPSIZE)
6        /sizeof(float);
7
8    // process set elements
9    for (int n=get_global_id(0); n<set_size; n+=get_global_size(0)) {
10
11     int offset = n - tid;
12     int nelems = MIN(OP_WARPSIZE,set_size-offset);
13
14     // copy data into shared memory, then into local
15     for (int m=0; m<4; m++)
16       arg_s[tid+m*nelems] = arg0[tid+m*nelems+offset*4];
17
18     for (int m=0; m<4; m++)
19       arg0_l[m] = arg_s[m+tid*4];
```

```
We see that tid = 0 in all cases, as get_local_id(0) always returns 0.
As a result, inside the for loop on line 8, offset = n and nelems = 4
or  2 (at the end of the set).

The problem starts being more clear on line 15, where the arg_s
indexes are given by tid+m*nelems, which are:
tid + m* nelems => 0 + m * 4 => [0, 4, 8, 12]

On line 18, the arg_s indexes are:
m + tid * 4 => m + 0*4 => [1, 2, 3, 4]

Here we clearly see that we allocate indexes 0, 4, 8, 12, but we
read from 1, 2, 3, 4, thus reading the wrong data.
```

Figure 4.1: Explanation as to why the code in Listing 4.4 can break when `OP_WARPSIZE` and the workgroup size are not the same.

Correcting `OP_WARPSIZE` has managed to fix all of the original Airfoil OpenCL codebase. All changes, including the thread divergence and synchronization were rendered unnecessary, given that we then learned OpenCL actually had workgroups with 1 workitem each.

Though fixed, the original codebase still had compatibility issues. The OpenCL changes were tightly integrated with the OP2 runtime, thus making the codebase unusable with any future versions of OP2. At this point, we started working on correcting its structure so that it uses the latest OP2 runtime and it complies with standards respected by all other OP2 Airfoil implementations (CUDA, OpenMP, etc.).

This process involved modifying some OP2 functions and mostly exporting all non-core OP2 functions in separate appropriate files.

Overall, the entire process of debugging the original OpenCL codebase took 5 weeks. This was due to our original assumptions about OpenCL and a lack of comments and proper design of the given code. Overall, this experience has taught us about OpenCL's limitations, differences between various OpenCL versions and given us a better understanding of how parameters vary based on architecture. Furthermore, we also learned what we can and cannot change with respect to the codebase, thus allowing us to reshape the project's scope.

4.1.2.1 OpenCL observations

Despite initially believing that any OpenCL code will give the exact same results when run on CPUs or GPUs, we have learned that this is not always the case. This very much depends on the code itself and, if not properly designed, there can be discrepancies between the code running on different platforms. This was shown by the usage of the `OP_WARPSIZE` variable, which prevented the code from giving right results when run on a CPU. Further differences have been observed when using an implementation that did not perform memory coalescing. This meant that, despite the fact that the code would run on a CPU, it would not be able to run at all on a GPU. From this we conclude that in order to achieve the best results when working with OpenCL, we need to design our programs based on a GPGPU design model.

Whilst testing the Airfoil with a number of OpenCL implementations on multiple platforms, we have observed that there are differences of both performance and correctness based on the combination of platform and OpenCL implementation. With optimizations turned on (`-cl-mad-enable` and `-cl-fast-relaxed-math`), on a Tesla M2030, both NVIDIA and AMD OpenCL gave similar performance. However, when running the Airfoil on a CPU, AMD with optimizations enabled ran in approximately 140 seconds, whilst the Intel one ran approximately 3 times faster, in 45 seconds. However, in this case, Intel OpenCL gave `NaN` results, whilst with optimizations off, it did not run any slower and gave the correct results.

Other differences include accepted syntax between NVIDIA, AMD and Intel. We have observed that Intel gave errors when it came to having `enum`

statements inside the kernels. We have thus adjusted the code appropriately.

For debugging purposes, we have also observed that there were differences between the level of features supported by various implementations. As NVIDIA OpenCL is strictly directed at GPUs, it has no support for `printf` statements, whilst AMD's and Intel's offer this feature. More specifically, the feature is the ability to use `printf` statements inside the computational kernels, which is very useful, as the entire computation happens at that stage and there is minimal information we can gather from the host code.

4.1.3 Compiler

The next step we needed to make in order to have a fully working set of tools was to fix the OP2 to OpenCL compiler. A major issue was that the compiler was not tested, therefore, we had no knowledge of the time required to fix it, nor of the necessary changes.

In order to find out what changes needed to be made, we used the compiler to generate OpenCL code and then we started fixing it. An initial problem that prevented the code from compiling was the fact that both the OpenCL and the host C++ code were generated in the same file. After splitting up the project, according to the original handmade OpenCL code (all kernel methods in one `.cl` file, and the rest into another a `.cpp` file containing the host code), we started to work on the correctness of the host methods. Some of the issues included improper usage of `clSetKernelArgument` with respect to buffers. As the name implies, this method sets the arguments for a given kernel. Other issues included improper or missing variable/pointer casting, or unallocated constants on device. Overall, the host functions contained minor errors.

Next to fix were the kernel functions and these had more complex errors. First of all, there were problems with constants that were not passed to the user supplied `inline` functions. Other problems included mismatched or missing address space qualifiers (also known as storage modifiers within the compiler's back-end). These are qualifiers that state in which memory region data associated with the variables is declared. One greater problem was a lack of qualifiers for formal parameters of the user defined function. This was due to the fact that the compiler's component that dealt with generating corresponding code had no knowledge of the kernel, and the formal parameter's variable names are different from the kernel's variables of the user defined function call. After exhausting options that relied on trying to define the address space qualifiers in the kernel function, we decided to use knowledge of the associated parallel loop parameters. We made one valid, working assumption that the order in which the parameters are stored is the

same as the order in which they are declared in the parallel loop.

Having solved this issue, we moved on to fixing casting and typing issues for shared memory objects. With respect to both kernels and user defined functions, we also solved issues around improper passing of arguments (example: all data arguments passed to the kernel have to be passed by reference).

At this point, we only had to solve two issues: correctly generating pointers that require more than one address space qualifier, and splitting of files.

The OP2 compiler is built with a tool called ROSE [34], which is an open source compiler infrastructure that can be used to generate source-to-source transformations. Despite officially having OpenCL support, ROSE does not offer a built-in option of generating declarations with two or more address space qualifiers. Therefore, we had two options to obtain this, both involving a hard-coded generation of the necessary output: we can either generate the test with `addTextForUnparser`, which is a string generated verbatim, or `buildOpaqueType`, which is a verbatim type. The advantage of the latter is that it is far easier to use and requires a lot less extra changes, as it can be given as a parameter to the declaration. However, the first option is far more complex to implement, as it affects the entire variable declaration and thus affects what we know of it. As a result, we chose to build the custom type with `buildOpaqueType`.

We then saw that generating more than one file within the compiler was not easily achievable and we needed extra scripts to deal with the process, as the compiler's back-end refuses to generate any files with extensions it does not recognize. Therefore, converting from a `.cpp` to a `.cl` file has to be done afterwards. As a result, we chose to write a Python script as shown in Appendix A.8 that splits the files appropriately. This method worked much better and allowed us to focus more on the machine learning algorithms instead of hacking the compiler.

For us to have usable generated code, we needed extra statements to be added to the generated code. These were lines of code that help calculating timing information, and a number of `#define` statements that allow us to control runtime parameters. Due to errors encountered with the compiler's back-end, for which all known workarounds failed, we resorted to adding all method calls through `addTextForUnparser`. For all other declarations and assignments, we used usual methods, which involved no hard-coding.

The overall process of fixing the compiler took an extra 5 weeks. Issues related to multiple address space qualifiers and their generation, splitting of files and generating appropriate storage modifiers were more difficult to debug and implement. Furthermore, problems with the compiler's back-end stalled our performance further as none of the workarounds available were successful, nor were the errors caused by code we added.

Nevertheless, we managed to have a fixed OP2 to OpenCL compiler, which now generates correct code for all variations of Airfoil it was tested with. However, as we have no other OP2 code examples available, due to the nature of converting C++ code into OpenCL code, there is a chance that the compiler will not generate working code. This is not due to compiler issues, but rather due to the fact that all the appropriate host code and kernel code, with the exception of user defined `inline` kernels, has to be generated from scratch. If the process is similar to all applications of this sort that use unstructured mesh applications, then the compiler will generate correct, working code.

4.2 Summary

In this chapter we talked about the technical difficulties that we have encountered in order to bring the necessary tools to a working state. We detailed on the process through which we fixed and improved the OP2 OpenCL runtime and described the methods we employed in fixing the OP2 to OpenCL compiler. In the end, we had both working OpenCL runtime and an OpenCL compiler which generated correct code. Furthermore, due to difficulties with the compiler's back-end, we chose to write a separate script which splits the resulting file which contains the host and kernel code.

Chapter 5

Machine learning techniques

In this chapter we discuss design and implementation choices of the machine learning techniques and how previous debugging experiences shaped the scope of the project. We present changes made to the scope of the project, additions made to the OP2 runtime and OP2 compiler, in shape of OP Tuner and a machine learning algorithm. We also describe the process we use for loop fusion evaluation and present a final script that achieves this.

5.1 Design choices discussion

When this project began, we envisaged it having a slightly different scope than the current one. An initial idea was to come up with a greater number of optimizations and choose machine learning algorithms to choose between them. However, this has changed when we realized the state of tools we wanted to use and the months required for getting them to work. Furthermore, some optimizations would have required a significant redesign of the OP2 runtime, changes that would have made the project's contributions unusable for anything else.

5.1.1 Choice of optimizations

We initially considered adding two main types of optimizations: implementation-based ones, such as AOS/SOA, and source-to-source ones, such as loop fusion and iteration reordering.

OP2 currently uses AOS and we believed it would be an interesting experiment to use SOA instead and see how it affects Airfoil's performance. There have always been debates and discussions as to when each way of storing data is preferred, and we believed that by using machine learning we

can more easily find a connection between a storage layout and an underlying architecture. This was possible as we had OP2 OpenCL to use, which works on a number of different back-end types (CPUs, GPUs, etc.). However, we soon realized that, despite being an interesting experiment, SOA was not a feasible option as it would have required major changes to the OP2 core runtime. As OP2 is used for multiple projects, within or outside of Imperial College London, any modifications brought to the OP2 core functions would break compatibility with all projects that use it. This would mean that our results would be isolated and unusable by anyone else. Furthermore, it would have also required significant changes to the OpenCL codebase. All of these reasons made us change our focus from implementation-based optimizations.

Iteration reordering was another optimization we considered as we believed it could improve performance, but we had no clear measure of its improvement. However, when we started discussing loop fusion, we experimentally observed a significant improvement in performance in other OP2 Airfoil implementations. Due to time constraints, we chose to go ahead with loop fusion, as we saw more clear benefits for using it on multiple codebases, as opposed to iteration reordering, which is more complex and might provide better performance.

When manually analysing the code, we realized that if we applied further transformations, we would be able to obtain extra loop fusions. Examples of this can be found in the Background chapter under code-block reordering example in Listing 2.5. Due to time constraints, we chose to only apply loop unrolling and code-block reordering manually, as they are not a core requirement of the project, and as long as we provide code examples with those optimizations enabled, we do not restrict our results' scope.

5.1.2 Scope of machine learning and deterministic automation

Initially, we aimed at using machine learning algorithms to determine an optimal and compatible combination of optimizations, however, the lack of options and time constraints have restricted this possibility. Furthermore, we also initially aimed to do all decisions whether optimizations were possible within machine learning. However, we later realized that such an attempt would only overcomplicate the algorithm and would not bring any benefit. Within this project we aim to show the impact of machine learning in conjunction with optimizing compilers, with respect to deciding on parameters and choice of optimizations based on a given architecture. We do not look to explore machine learning techniques' full capabilities. As a result, we decided to determine certain properties automatically, outside of machine learning.

More specifically, we are determining which loops are fusable, by using basic control flow information, automatically. We pass the possibilities onto the machine learning algorithm that further generates an optimal choice of loop fusions.

In addition to which fusions we should perform, the machine learning algorithm, based on a given architecture, decides on values for runtime parameters. We chose to have those, as different values render variations in performance, which is what we aim to optimize.

Due to time constraints, we only chose to implement and offer support for loop fusion and runtime parameter tweaking, whilst code-block reordering and loop unrolling are performed manually. This choice does not limit the experimental side of the project, nor its scope. We still have a clear proof of concept by using this combination of optimizations. By using the tools we improved for this project, we can extend this work in a great number of ways, as we only provide the first step towards architecture-based optimization choices. Further details of extension possibilities are discussed in Chapter 8: Conclusions, under Further Work 8.2.

5.2 OP2 Runtime support – OP Tuner

At this stage, all the tools we chose to use are fixed. As a result, we first implemented the runtime support for the machine learning algorithm. This is minimalistic, as we have no runtime optimizations. We have tweaking of runtime parameters, which is done through the runtime support. Furthermore, this allows manual manipulation of parametefrs, from within Airfoil. Despite having a minimal effect at the moment, the runtime support can be extended further, depending on the choice of optimizations we have.

We chose to create a C structure called `op_tuner` which can be declared within the `airfoil.cpp` file, which then controls the runtime parameters of `BLOCK_SIZE`, `PART_SIZE` and `OP_WARPSIZE`. `BLOCK_SIZE` represents the number of threads in a workgroup, from the OP2 plan function's perspective, `OP_WARPSIZE` represents the number of threads executing in lockstep inside the kernel, whilst `PART_SIZE` shows the number of elements in a partition. Due to the changed scope of the project, the `op_tuner` structure has fields which are no longer used. Listing 5.1 shows the structure of `op_tuner`.

Listing 5.1: Here we show the `op_tuner` structure, and associated helper elements. As we can see, it contains the block, partition and warp size. The cache line size was originally considered, but it did not affect the code as much as the others, thus was not used in the final implementation. The structure also contains a name, an active flag, and whether it is a loop or global tuner. The architecture shows on which architecture the code is running. Currently, this functionality is not enabled, as we do not have any runtime optimizations.

```
/*
 * The core of the runtime op_tuner
 */

typedef enum {ANY, CPU, GPU, ACCELERATOR} arch;

typedef struct {
  int   block_size;
  int   part_size;
  int   cache_line_size;
  int   op_warpsize;
  arch  architecture;
  char const *name;
  int   loop_tuner;
  int   active;
} op_tuner;

struct node {
  op_tuner *OP_tuner;
  struct node * next;
};
```

From an implementation perspective, in order to ensure the correctness of the `op_tuner` declaration, we have created two functions called `op_create_global_tuner()` and `op_create_tuner("tuner_name")`. This is to ensure that all fields are properly initialized as, for example, having a `PART_SIZE = 0` will lead to erroneous results. Furthermore, we have two tuner declaration functions, as our original design involved having two types of control over the program parameters: one involving general changes that apply to the entire code, and the second involved changes that apply only to particular `op_par_loops`. As an example, consider the case of choosing between AOS and SOA. If we had this optimization, we would have to apply it throughout the entire program, thus it would be a feature of the global tuner. However, choosing whether we want a particular loop to have a different partition size compared to others is a change that only affects that particular loop, thus it would be controller by loop tuners. This approach ensures that loop tuners cannot overwrite the global tuner, and thus maintain optimization

correctness and consistency throughout the program. However, as the scope of the project has changed, the role of the global and loop tuners has adapted as well. We maintain a possibility for them to fulfill their intended purpose, nevertheless, at the moment, they only control a few parameters, and the global tuner ones can be overwritten by loop parameters, as long they are correct. A correct parameter must be non-zero or non-null. The same rule applies to the global tuner. If the user overwrote the default values with improper ones, the user default values would be discarded when we attempt to use them, and they are substituted with default ones.

Further helper methods have been defined, such as `op_get_global_tuner()` and `op_get_tuner("tuner_name")`, for retrieving global and loop tuners, respectively. A full list of methods can be found in Appendix A.2: OP2 Tuner Runtime Support.

After further consideration, we realized that the only feature the current implementation of `op_tuner` provided was to overwrite runtime parameters. The same result can be achieved through normal `#define` statements, which can be passed values from a Makefile. As a result, we chose not to implement the `op_tuner` support into the OP2 compiler. However, runtime and script support remain. When future runtime optimizations are added, providing compiler support for the `op_tuner` will be more justified.

5.3 OP2 Compiler support

Within the OP2 compiler, we added timing information which tells us how long each kernel took to execute, thus giving us a more detailed picture of our optimization's impacts. As mentioned in Section 5.2 OP2 Runtime support - OP Tuner, we chose not to add compiler support for `op_tuner`. In exchange, we chose to add similar `#define` statements, which achieve the same result. Without these, we would not have been able to control any parameters at runtime, with the exception of `OP_WARPSIZE`, which is overwritten at a different stage during runtime.

Inserting timing information, despite initially appearing to be a simple task, proved to be much more challenging. This is due to errors of the compiler's back-end, which started throwing errors when adding any new method calls. No workarounds managed to fix the issue, and we ended up generating the desired code through `addTextForUnparser`, thus hard-coding the method calls. This is not an issue, as these are the same across programs. All other declarations and assignments are generated appropriately, without any hard-coding of output.

Initially, the OP2 compiler had no loop-fusion support. As a result, we

added this optimization. The fusion is performed in a simplistic way, in which it just appends the user kernels, and the parameters that are passed to the loops. There are no checks whether the fusion is possible or not. This is achieved by automation, as detailed in Section 5.4: Loop fusion evaluation.

5.4 Loop fusion evaluation

The fusion implementation in the OP2 compiler has no knowledge of whether two loops can or cannot be fused. Therefore, in order to successfully use this feature, we had to introduce loop fusion evaluation. The evaluation is comprised of a small control-flow information analysis that is applied to `airfoil.cpp`. It creates scoping information for `for` and `op_par_loop` loops. If two `op_par_loops` are in the same scope and are next to each other, they they can be fused. This algorithm does not consider whether we should or should not fuse them, but only if the fusion is possible. We achieve this by creating a tree, which has the following node structure 5.2:

Listing 5.2: This is the Root node of the Control-flow information analysis. The analysis is small, so we store the name of the node, its children and depth. This information is sufficient to analyse whether loops are fusable or not.

```
CFInfoRootNode = {
  'name' : 'ROOT',
  'children' : [],
  'depth' : 0
}
```

The Root node is the only one with `depth = 0`, and all other statements have a `depth >= 1`. We add each statement as a node, in the order we find them, and in the list of `children`, from left to right. This means that we preserve information of order of statements within the code. Furthermore, only `for` loops can generate a new scope, thus a new level of nesting. With respect to naming, the root node is called "ROOT", however, all other nodes have more relevant names. `for` loops have a name with the following format: "for" + nth `for` loop index, whilst `op_par_loops` keep the name of the loop to which we add an mth `op_par_loop` index. We call this indexed `op_par_loop` name a tag. As we can have multiple occurrences of an `op_par_loop`, we add the extra index. This helps us properly identify each occurrence within our control flow information tree, thus generating a correct analysis of which loops are fusable, and which are not. In order to maintain knowledge of which loop matched which tag, whilst generating the tree, we create a map of tags and original loop names. An example of map can be seen in Listing 5.3. As a result, we obtain a tree of the form shown in Listing 5.4:

Listing 5.3: This is an example of the tag to original name map that we create. Its actual data structure is a dictionary.

```
[{'tag': 'save_soln0', 'original': 'save_soln'},
 {'tag': 'adt_calc1', 'original': 'adt_calc'},
 {'tag': 'res_calc2', 'original': 'res_calc'},
 {'tag': 'bres_calc3', 'original': 'bres_calc'},
 {'tag': 'update4', 'original': 'update'}]
```

Listing 5.4: This is a full Control-flow information tree, for the standard Airfoil code. From here, we can see that the only fusable loops are `adt_calc` and `res_calc`, `res_calc` and `bres_calc`, and `bres_calc` and `update`.

```
{'depth': 0, 'name': 'ROOT', 'children':
  [{'depth': 1, 'name': 'for0', 'children':
    [{'depth': 2, 'name': 'save_soln0', 'children': []},
     {'depth': 2, 'name': 'for1', 'children':
      [{'depth': 3, 'name': 'adt_calc1', 'children': []},
       {'depth': 3, 'name': 'res_calc2', 'children': []},
       {'depth': 3, 'name': 'bres_calc3', 'children': []},
       {'depth': 3, 'name': 'update4', 'children': []}
      ]}
    ]},
    {'depth': 1, 'name': 'for2', 'children': []}
  ]}
```

At this point, we can generate a list of fusable loops, by first creating a list of available `op_par_loops`, as shown in Listing 5.5, and then traversing the newly formed list, checking in the control-flow information tree that the two loops we are considering are found within the same scope (i.e. have the same depth), and that they are found next to each other (by being in the parent's `children` list next to each other. At the end, we use the map of tags and original names to generate a list of fusable loops that have their original names.

Listing 5.5: List of available loops, in order of appearance.

```
['save_soln0', 'adt_calc1', 'res_calc2',
 'bres_calc3', 'update4', 'save_soln0']
```

In Listing 5.6 we can see a list of fusable pairs, where each pair contains the names of each of the loops involved in the loop fusion.

Listing 5.6: List of possible loop fusions.

```
[{'loop2': 'res_calc', 'loop1': 'adt_calc'},
```

```
{'loop2': 'bres_calc', 'loop1': 'res_calc'},
{'loop2': 'update', 'loop1': 'bres_calc'}]
```

This result is then passed on to the machine learning algorithm, which then decides which of the fusions is worth performing.

5.5 Machine Learning

Our implementation's final component is the machine learning algorithm, which provides decisions of which loop fusions to perform and what values we should choose for the runtime parameters. We implemented a case-based reasoning system, which uses a structure outlined in Figure 2.6. As a result, the system has the following main components:

1. CBRInit – a method used for initializing the system with the training cases and their respective outcomes. It outputs a resulting CBRSystem.

2. retrieve – this method retrieves a case most similar to our unsolved one, and returns it. This similar case comes from its case-base.

3. reuse – once we found a best matching case, we want to apply its solution to our new case. This method does just that.

4. retain – the last step in this process is to retain the newly solved case in the system's case-base.

We represent the CBRSystem as a list of CBRSystemCases, which are made out of a CBRCase, CBRSolution and a number of occurrences. Each CBRCase contains a target architecture, a list of fusable pairs, as dictated by the loop fusion evaluation algorithm, and a list of all `op_par_loops` available. A CBRSolution is made out of a list of loops which we are going to fuse and the `op_warpsize`, `block_size` and `part_size` values. In Listing 5.7, we present this data structure's implementation.

Listing 5.7: Here we show how we have implemented the base components of the case-based reasoning system.

```
#  All these dictionaries are initialized with stock values,
#  as they are only listed for reference.
CBRCase = {
          'arch' : Arch.ANY,
          'fusable_pairs' : [],
          'op_par_loops' : []
          }
CBRSolution = {
```

```
                'loops_to_fuse' : [],
                'op_warpsize' : [],
                'block_size' : [],
                'part_size' : []
                }
CBRSystemCase = {
                  'case' : CBRCase,
                  'solution' : CBRSolution,
                  'occurrences' : 1
                  }

# The CBRSystem is initially empty, and values are assigned
# in the CBRInit stage.
CBRSystem = []
```

CBRCase is what we initially create when we have an unsolved problem. It is initialized as in Listing 5.8, where we show that the architecture takes a `default_arch` value, which can be overwritten by the user through a script parameter. `Fusable_pairs` and `op_par_loops` are initialized by the variable called `fusable_pairs`, and `op_par_loops` respectively. The former is given appropriate values at the loop fusion evaluation stage, whilst the latter is initialized when we parse `airfoil.cpp`. Afterwards, we create a CBRSystemCase, which is then passed on to all other CBR methods as an unsolved case, by referencing the newly created CBRCase as its `case`, and giving it an empty `solution`, and one occurrence.

Listing 5.8: Initialization of unsolved case, which is passed onto the CBR system.

```
# initial unsolved case for the CBR system
newCase = {
          'case' :  {
                    'arch' : default_arch,
                    'fusable_pairs' : fusable_pairs,
                    'op_par_loops' : op_par_loops
                    },
          'solution' : None,
          'occurrences' : 1
          }
```

Once we have an unsolved case, we pass it on to the `retrieve` method, which gives us a best matching case, and then to `reuse`. During `retrieve`, we choose to validate the results, by using a `checkBestMatch` method. It can happen that our current case is something previously unseen, and therefore, despite the fact that the CBR system might work appropriately, its lack of knowledge with respect to this case might produce a sub-optimal result. Furthermore, it can happen that our new case belongs to a different program

and the loops do not match. As a result, finding a connection between the cases would prove far too difficult and error-prone, thus we chose to generate a new solution based on the new case by performing code analysis. Whilst coding the CBR system, we observed that determining which loop fusions are optimal and what values runtime parameters should have can be a fully deterministic process. Therefore, `checkBestMatch` verifies whether the machine learning algorithm made any obvious mistakes and tries finding a better option. If none is available, we return the one suggested by the CBR system. Otherwise, we create a new option. This is our fail-safe. More about fail-safes can be found in Section 5.5.3: Fail-safes.

The final solution contains values for `block_size`, `part_size`, `op_warpsize` and a list of feasible fusions. Its full structure in shown in Listing 5.9.

Listing 5.9: A possible solution, returned by the CBR system as part of a solved case.

```
# a potential solution, returned as part of a solved case
CBRSolution =  {
                'loops_to_fuse' : [],
                'op_warpsize' : 1,
                'block_size' : 32,
                'part_size' : 256
                }
```

Due to the nature of the OP2 Compiler's fusion, we can only provide it with one fusion at a time. In order to maintain flexibility, we only return the best fusion. In order to obtain all fusions, we would need to rerun each result through the tuning algorithm until we get no more fusions. Furthermore, by only returning one fusion and not choosing to suggest all possible fusions, we keep in consideration situations when the freshly fused loops can be fused with a 3rd loop, which would, in the initial step, be involved in a different fusion.

5.5.1 Similarity measure

In order to determine which is the most similar case to our unsolved one, we first check whether we have an identical one stored in the CBR system. If not, we employ a particular implementation of the k-nearest neighbour algorithm, which gives us the closest match. This is done through a method called `similarityEstimation`. Here is the first occurrence where we consider the unsolved case as a vector of properties. The first four entries in the vector correspond each to a target architecture, whilst the fifth corresponds to the fusable loops weighting, and the sixth to complexity of the given loops, with respect to loop fusion. A visualisation of this is available in Figure 5.1. A

detailed description of used weighting and complexity algorithms can be found in Section 5.5.2: Weighting and complexity calculations.

Arch.ANY	Arch.CPU	Arch.GPU	Arch.ACC	Loop Fusions Weights	Loop complexity

Figure 5.1: Vector of properties for the CBR system, where the first 4 entries match the 4 types of supported architectures (ANY, GPU, CPU, ACCELERATOR), whilst the fifth entry represents the weighting of fusable loops, whilst the sixth has the complexity of the loops, calculated based on possible loop fusions.

The method then calculates, for each case in the CBRSystem, a weighting for its individual parameters. This weighting is directly proportional to the number of occurrences for the respective case. Once this is achieved, we normalize the results, to exclude errors that can be introduced by frequency of cases.

Once this is done, we calculate the weighting of intersection between our unsolved case and each existing compatible case, taking in consideration the weight of each property, which we calculated at the previous step. A compatible case is one that has the exact same `op_par_loops` as the unsolved one. We take the maximum of all these weights and then we add all cases that have a maximum weight to a new list called `maxList`. This is subsequently sorted based on the occurrences of each case. If `maxList` is empty, then we return `None`. The `maxList` can be empty when our new case's `op_par_loops` differ those of every other known case.

5.5.2 Weighting and complexity calculations

Within the `similarityEstimation`, we need to associate weighting to each vector parameter. We calculate those as follows:

- The matching architecture takes a weighting of +5.
- The fusable loops is the total number of fusable loop pairs that are roughly worth fusing. A pair of loops is roughly worth fusing if they traverse the same set. If the set differs, then the fusion is certainly not feasible.
- The weighting of loop arguments is calculated based on the total complexity given by possible loop fusions. In turn, this is a function of the

number of operations of the user supplied kernel, per loop, and the number of equal, similar, different and total arguments. We calculate the number of operations per loop as follows: each mathematical function call, operation and assignments are each worth 1 point, `if` statements are worth 2 points, whilst `for` loops are worth 3. We chose these weights, as we abstract away the complexity of mathematical calls, operations and assignments and we consider them as one operation. However, `if` statements contain a comparison and a jump, which involves two operations. Similarly, `for` loops contain one comparison, one increment and a jump, totalling to 3 operations. Two arguments are considered equal is they share the same data array, data array access and map. Two arguments are similar if they have the same data array and map, but different access. They are different in all other cases. Furthermore, we do not care about the type of access of the data, as we are confirmed to do all operations in appropriate order. The complexity of a loop fusion is given by the sum of:

- overall amount of operations, multiplied by the weighting associated with the number of operations
- proportions of equal arguments, multiplied by the weighting of equal arguments and the number of equal arguments
- proportions of similar arguments, multiplied by the weighting and then number of similar arguments
- the total proportion of different arguments, multiplied by the weighting and total number of different arguments, respectively
- total number of arguments, multiplied by its respective weighting

Each of the weightings is multiplied by the respective case's number of occurrences. Furthermore, all hard-coded values we use have been decided based on experimental observations, and they depend on the architecture. We currently have a set for CPUs and one for the rest.

Whilst developing the formulas for these weightings, we observed that there is a deterministic way to calculate the desired loop fusions and the values for the runtime arguments. The loop fusion complexity calculation is one representation of such a deterministic method. The method is further described in Section 5.5.3: Fail-safes.

5.5.3 Fail-safes

Fail-safes are fully deterministic methods that can give us optimal or close to optimal results, with respect to a set of parameters and optimizations, for

a given program. At the moment, we have only one fail-safe implemented, which gives us a better solution in case we do not have any matches of the current case in the case base, or if the current best case has clear errors. A clear error is, for example, a `OP_WARPSIZE` different than 1 for a CPU solution.

The fail-safe calculates the maximum complexity, from all the loop fusion complexities within the fusable pairs of the new case. The fusion with the maximum complexity is the desired one. If the best match contains the wanted fusion and is comparable, then it is a good case. Otherwise, assuming the best match has extra knowledge that we do not have, and perhaps the wanted fusion is not as desirable, we check whether the `OP_WARPSIZE` is 1 if the architecture is CPU. If this is the case, we return the best match, otherwise we calculate our own solution.

We first set the `OP_WARPSIZE` to 1 if the architecture is anything but GPU, which gets a value of 32. We then use the complexity calculation of each loop's arguments, and evaluate the maximum one. We calculate this by first determining the total number of similar arguments within the same loop, and we make an array of distinct such accesses. We multiply their total number by a predetermined weight, which depends on the architecture. Then, we add to the complexity the total number of arguments the loop has, multiplied by its respective weighting. We return the result. Just as in the case of all other weights discussed, the hard-coded values have been determined experimentally.

Afterwards, we calculate appropriate values for block and partition sizes. We start from the standard value of 128 and, depending on the architecture, we compute the block size as a function of the overall maximum complexity, a reference value of 4 and a custom adjustment factor, architecture dependent. It is calculated as follows:

```
math.log(overallMaxComplexity * referenceValue) *
                adjustmentFactorArch
```

This gives us a temporary size, which we then compare with the possible values for block size, which are powers of 2 starting from 4 and ending with 512. We choose the value that is closest to ours. The partition size depends on the value of the block size. As a result, we use a custom defined threshold value which dictates the ratio between partition size and block size. As we observed experimentally, the ideal case is when the partition size is 8 times the block size, therefore, we aim to keep that ratio. If it is not possible due to a too high block size, we decrease the ratio in powers of two until we are able to fit an appropriate value up to 512 for the partition size.

Once we have all this data, we can put together a new solution for our problem.

A potential second fail-safe would be to run the compiler at the end of the machine learning algorithm, in order to determine whether our result is good or not. Partial support exists for this, however, due to manual tweaking needed to run the loop fusion option within the compiler, we have decided that it is better to tweak the hard-coded values manually, based on experimental results.

5.5.4 Overtraining

Due to our very small case-base, we are very likely over-fitting the algorithm for Airfoil. The fail-safe we use also enforces over-fitting, as we are introducing cases that are custom tailored to a specific problem. Nevertheless, once more cases are added to the system, we will safely be able to disable the fail-safes or, even if we choose to leave them in place, we can at least not add their results to the CBR system. At the moment, we use this last option of not storing the results. Despite having support for adding them to the CBR system, we do not actually save them outside of the script, and the script runs for only one case. As a result, we limit the amount of data overfitting that occurs and encourage a more thorough testing of the deterministic algorithms through unseen cases. If we were to add and store permanently all cases, we would severely overfit the machine learning system and we would also cover all possible cases quickly enough, thus turning our machine learning problem into a simple lookup.

5.5.5 Training data quality

In order to thoroughly test the full algorithm, we only added test cases that refer to the stock Airfoil code. For any fused versions, our system uses the deterministic approach to deliver a good result. We chose this method, as it encourages us to fine-tune the system appropriately, and it provides a more extensive testing.

Furthermore, the data itself is optimal or close to optimal, and the results were experimentally determined, after running Airfoil under a number of configurations.

5.5.6 Validation of results

As our second fail-safe is not currently activated, we determine whether a result is optimal or close to optimal by running experiments which involve tweaking of parameters. Furthermore, we know that by moving further away from the ideal parameter values, in either direction (i.e. making them too

small or too large) will continuously decrease performance, it is safe for us to try a small number of configurations. Furthermore, we compare the result against the stock Airfoil code, which we use as a benchmark for all our optimizations.

A more detailed discussion, including execution time values and comparisons of experiments, Airfoil variations and architectures is offered in Chapter 6: Evaluation.

5.6 Script

For ease of use and simplicity, we incorporated the text parsing, control flow information, machine learning and deterministic algorithms, and even functionality to call the compiler into one Python script, called `tuner.py`. The purpose of this script is to be run with parameters for target architecture, a file to be parsed and the number of operations of associated user kernels, and all the operations are performed seamlessly. A comprehensive version of the script can be found in Appendix A.4: Tuner script.

5.7 Summary

In this chapter we provided a discussion of our choice of optimizations, and detailed how the scope of the project adapted during time. We then went on and presented the work we performed on existing tools, including improving the OpenCL runtime and fixing the OpenCL compiler. After that, we talked about the newly added OP2 tuning runtime and compiler support. Finally, we detailed on the machine learning techniques we used, and how they fused with deterministic algorithms which help us provide more reliable better results.

Chapter 6

Evaluation

The main purpose of this project is to investigate whether machine learning and deterministic algorithms applied to OP2 code can give a performance improvement, in the shape of better execution times for unstructured mesh applications.

In this chapter we provide experimental results that show how the suggestions of the machine learning and deterministic algorithm perform against the stock version of Airfoil. We also include results from other variations of Airfoil, which contain non-optimal fusions, that were used to tweak the algorithm's weightings, as detailed in Section 5.5.2: Weighting and complexity calculations. All experiments were run on two different configurations, both having CPUs as main compute devices.

6.1 Testing platforms

To ensure our results are consistent across machines and architectures, we used two different compute configurations for all tests. They are as follows:

- Imperial HPC center machine: Intel Xeon X5650 2.67 GHz with 24 GB RAM, having 6 cores and Hyper Threading, totalling 12 threads
- Personal machine: Intel Core i7 2620M 2.7 GHz with 4 GB RAM, having 2 cores and Hyper Threading, totalling 4 threads

The OpenCL runtimes we used were Intel OpenCL 2.0 on the Xeon and 1.5 on the Core i7. For these tests we used no GPUs, as we encountered various OpenCL runtime issues when trying to test the software. These issues are likely caused by NVIDIA compiler bugs, which we cannot fix. We did not use AMD's platform, as it offers OpenCL GPU support only for its own

graphics cards, and the cards we had available were NVIDIA Tesla M2050. Furthermore, we also rejected AMD's OpenCL as, from past experience, it was significantly slower than Intel's (approximately 2.5–3 times), and with the latest version of the generated code, it showed inconsistent behaviour across configurations, despite being the same version (AMD OpenCL 2.6). Whilst on the i7 it failed to build, on the Xeon it failed to run due to semaphore wait issues.

Despite using only two CPU-based configurations with their respective Intel platforms, we successfully show performance results of fused Airfoil versions against the stock one.

6.2 Results for Airfoil variations

As we wanted to create algorithms that render performance improvements, we first needed to properly understand Airfoil, the architectures and the available optimizations. We started from the assumption that all fusions will render better execution times. As a result, the first set of experiments we ran involved versions of Airfoil which performed all possible feasible fusions. We consider a feasible fusion to be one which involves two loops which iterate over the same set. If they iterate over different sets, we get no performance benefits, as we do not have any shared data between the loops.

In Airfoil, three pairs of loops are fusable: `save_soln` and `adt_calc`, `adt_calc` and `update`, and `update` and `save_soln`. We provide two variations of Airfoil, in addition to the stock one. The first performs loop unrolling and includes the first two possible fusions, whilst the second performs loop unrolling and code block reordering, thus enabling the last two possible fusions. A visualisation of this is provided in Listings 6.1 and 6.2.

Listing 6.1: Airfoil variation with loop unrolling and the following fusions: `save_soln` and `adt_calc`, and `adt_calc` and `update`.

```
1
2  n = 1000;
3
4  for (int i = 0; i < n; ++i) {
5      op_par_loop("fused_save_soln_adt_calc");
6      op_par_loop("res_calc");
7      op_par_loop("bres_calc");
8      op_par_loop("fused_update_adt_calc");
9      op_par_loop("res_calc");
10     op_par_loop("bres_calc");
11     op_par_loop("update");
12 }
```

Listing 6.2: Airfoil variation with code-block reordering and loop unrolling, and the following fusions: `adt_calc` and `update`, and `update` and `save_soln`.

```
1
2  n = 1000;
3
4  op_par_loop("save_soln");
5
6  for (int i = 0; i < n-1; ++i) {
7      op_par_loop("adt_calc");
8      op_par_loop("res_calc");
9      op_par_loop("bres_calc");
10     op_par_loop("fused_update_adt_calc");
11     op_par_loop("res_calc");
12     op_par_loop("bres_calc");
13     op_par_loop("fused_update_save_soln");
14 }
15
16 op_par_loop("adt_calc");
17 op_par_loop("res_calc");
18 op_par_loop("bres_calc");
19 op_par_loop("fused_update_adt_calc");
20 op_par_loop("res_calc");
21 op_par_loop("bres_calc");
22 op_par_loop("update");
```

For all these tests, we varied the block and partition sizes between 8 and 512, giving them powers of two values and making sure that partition size is at least as high as the block size. We show how execution time varies depending on these parameters for the original codebase and the two fusion versions in Figures 6.1, 6.2 and 6.3, respectively, for the Xeon results, and Figures 6.4, 6.5 and 6.6, respectively, for the Core i7 ones.

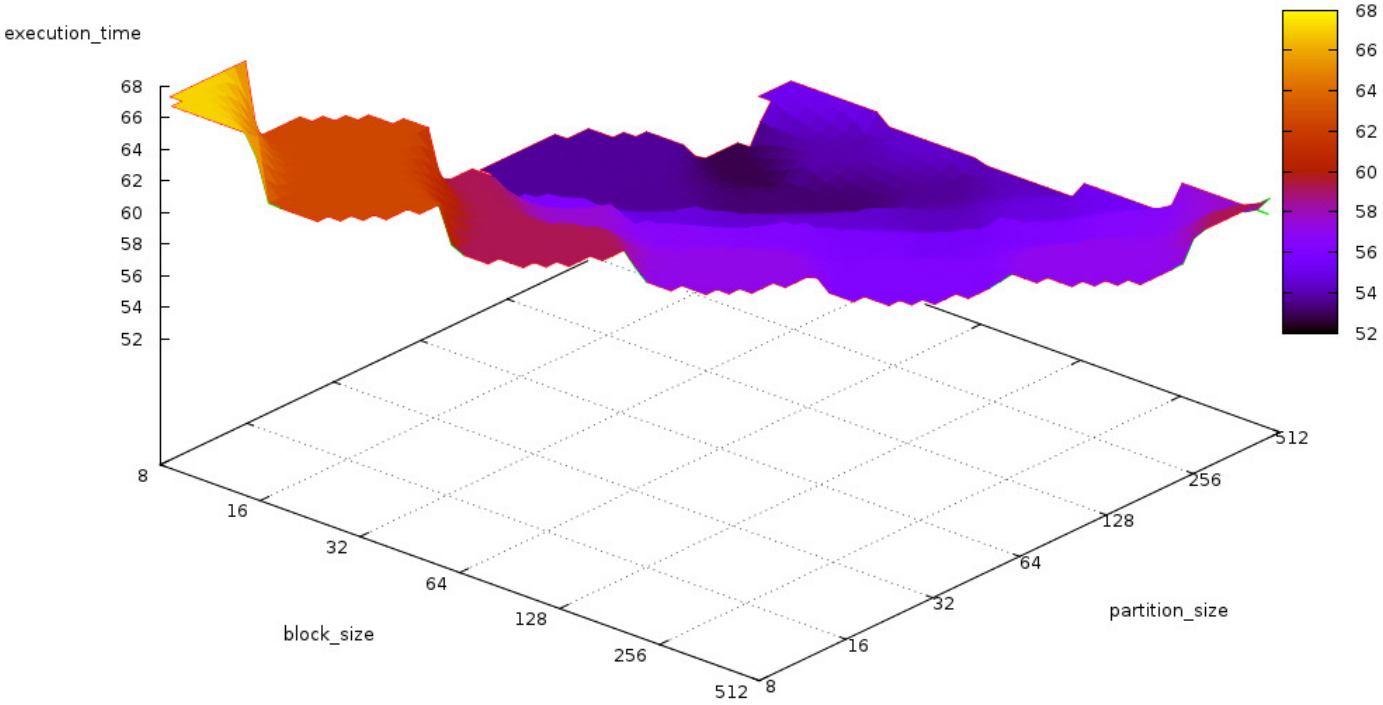


Figure 6.1: Xeon results for the original Airfoil codebase. These show that as long as we maintain a relatively small block size (such as 16 or 32) and we increase the overall partition size, gradually, our execution time is going to drop from 68 seconds to 52 seconds.

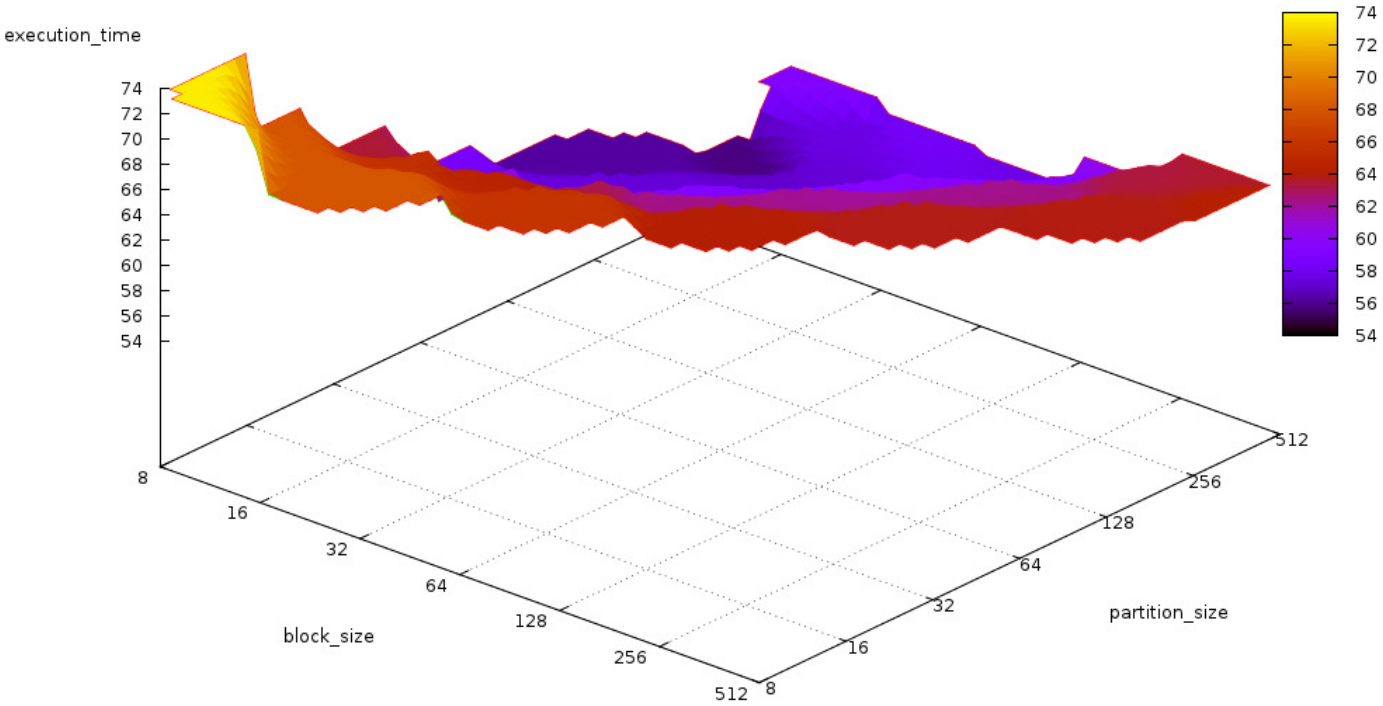


Figure 6.2: Xeon results for Airfoil with fused `save_soln` and `adt_calc`, and `adt_calc` and `update`. Here we can see that the performance increases (from 74 seconds down to 54 seconds) as long as we maintain a proportionally inverse relationship between the block size and partition size, and we increase the partition size.

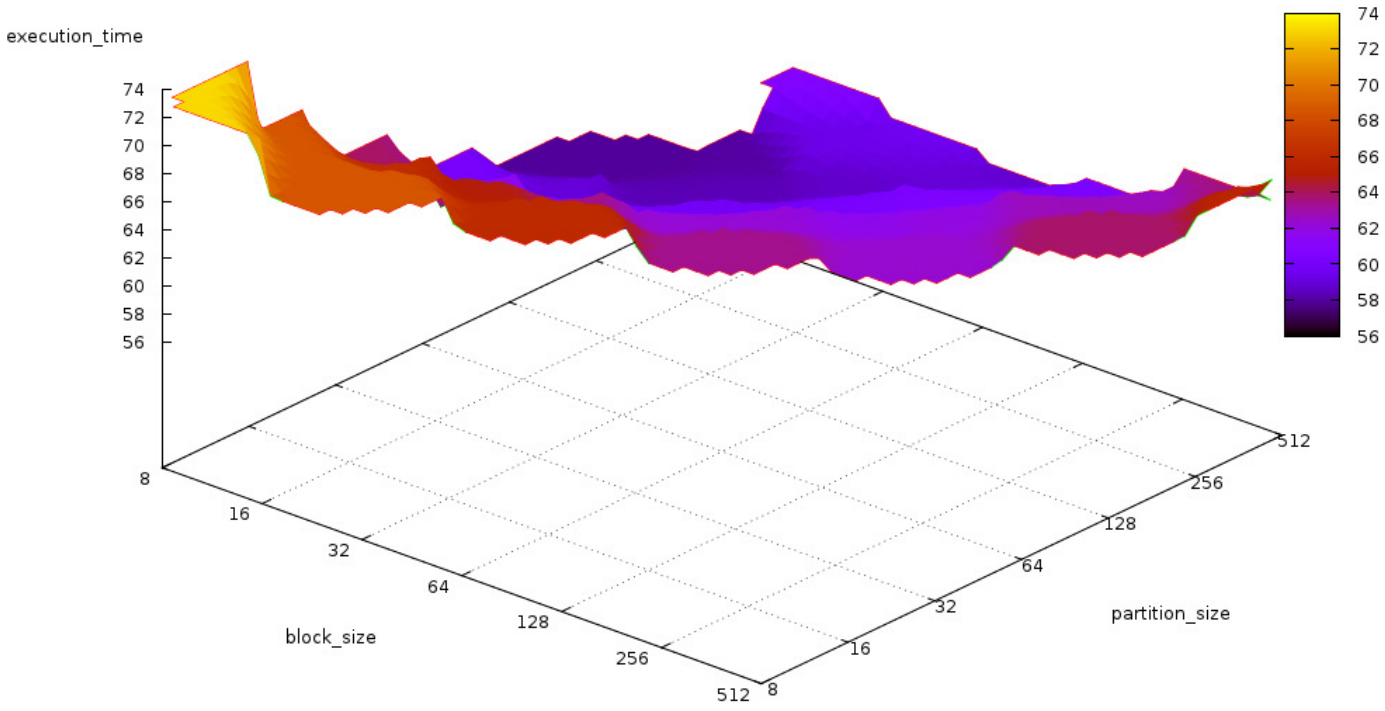


Figure 6.3: Xeon results for Airfoil with fused `adt_calc` and `update`, and `save_soln` and `update`. We observe that we can reduce execution time from 74 seconds down to 56 seconds, by keeping the block size between 16 and 32, whilst increasing the partition size up to 256.

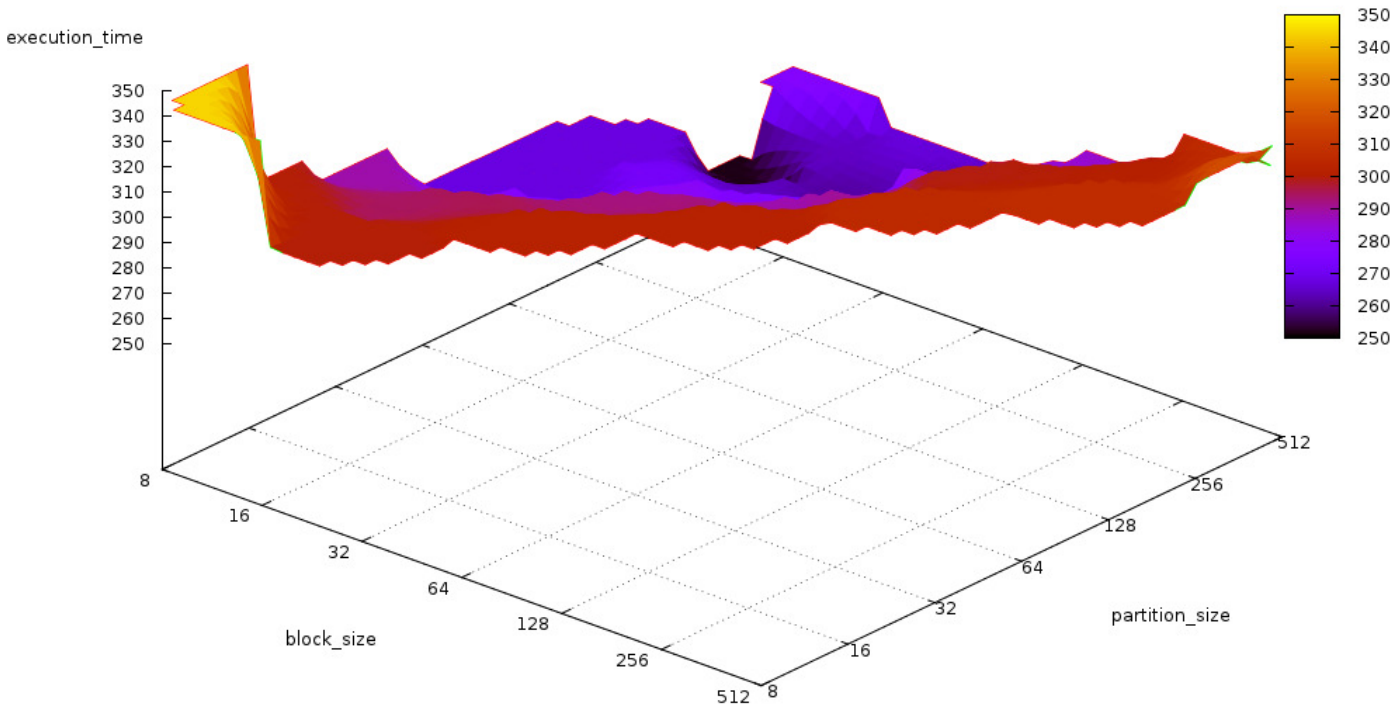


Figure 6.4: Core i7 results for the original Airfoil codebase. Just like in the Xeon results, we observe that we can improve execution times by 100 seconds, by keeping a small block size and increasing the partition size.

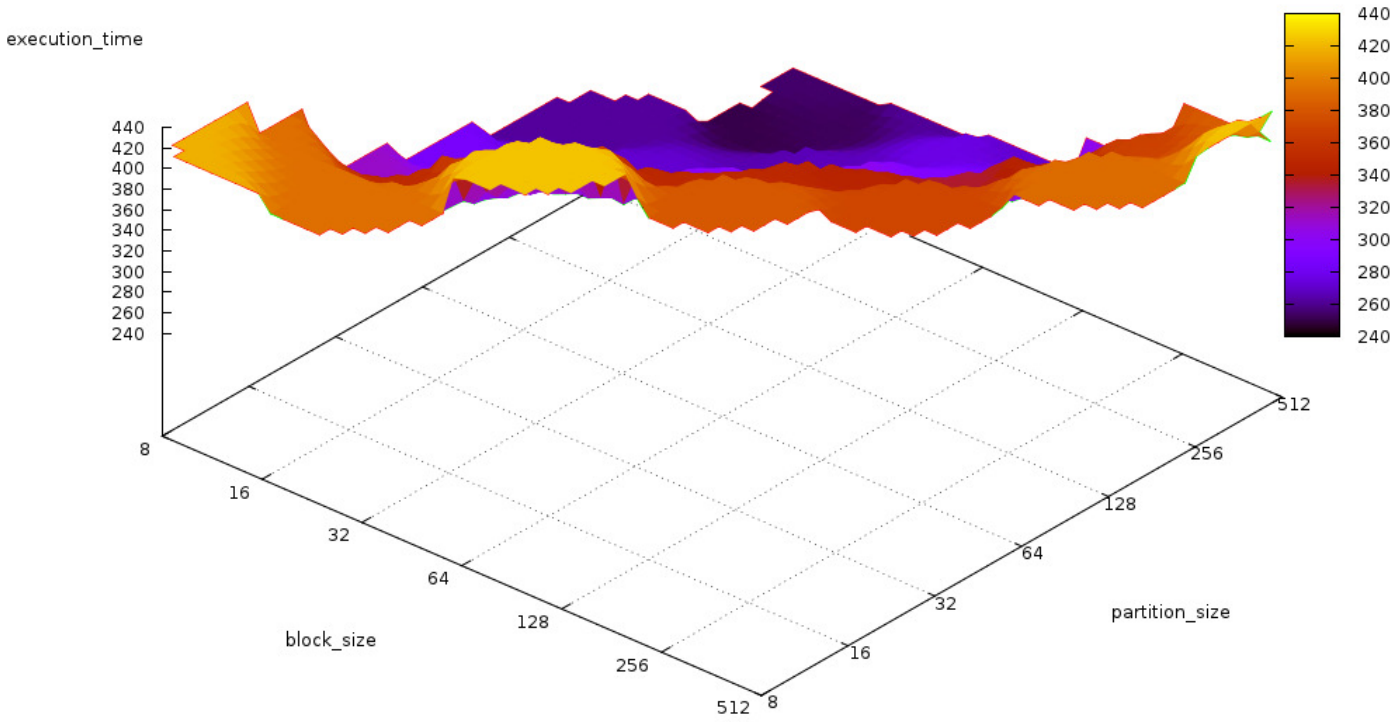


Figure 6.5: Core i7 results for Airfoil with fused `save_soln` and `adt_calc`, and `adt_calc` and `update`. Here we see that we can obtain significant performance improvements of up to 200 seconds by using block sizes of 16–64 and partition sizes of 256–512.

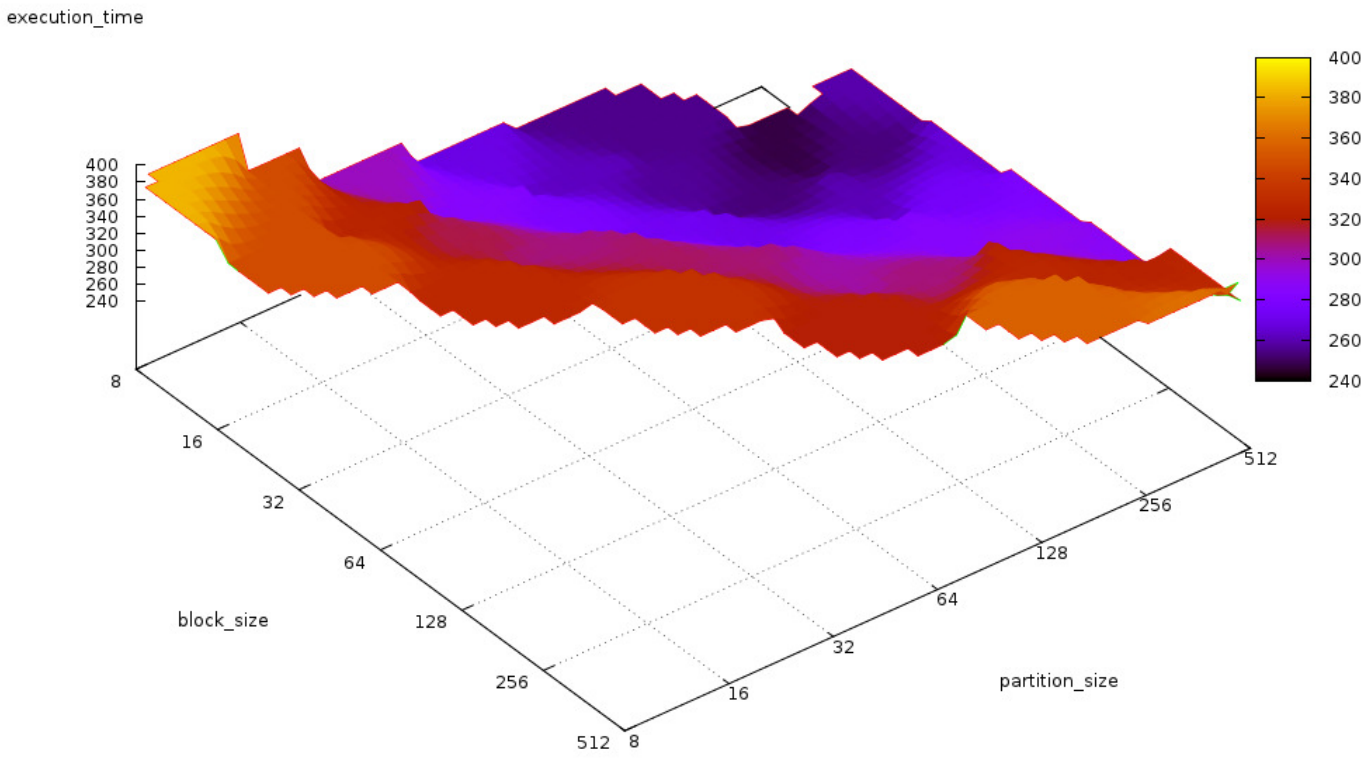


Figure 6.6: Core i7 results for Airfoil with fused `save_soln` and `adt_calc`, and `adt_calc` and `update`. This shows us that by using a block size of 32 and a partition size of 256, we can reduce the execution time from 400 seconds (in the case of block size and partition size of 8) down to 240 seconds.

As we can observe in Figures 6.1–6.6, irrespective of which configuration and Airfoil version we have, there is an optimal pair of block and partition size $(32, 256)$. At this point we can test this further by enabling one fusion at a time and seeing whether it produces optimal execution times. To ensure that these values are optimal, we ran experiments with the following neighbouring pairs: $(16, 128)$ and $(32, 512)$. We show the results in Figures 6.7 and 6.8 for the Xeon and Core i7, respectively.

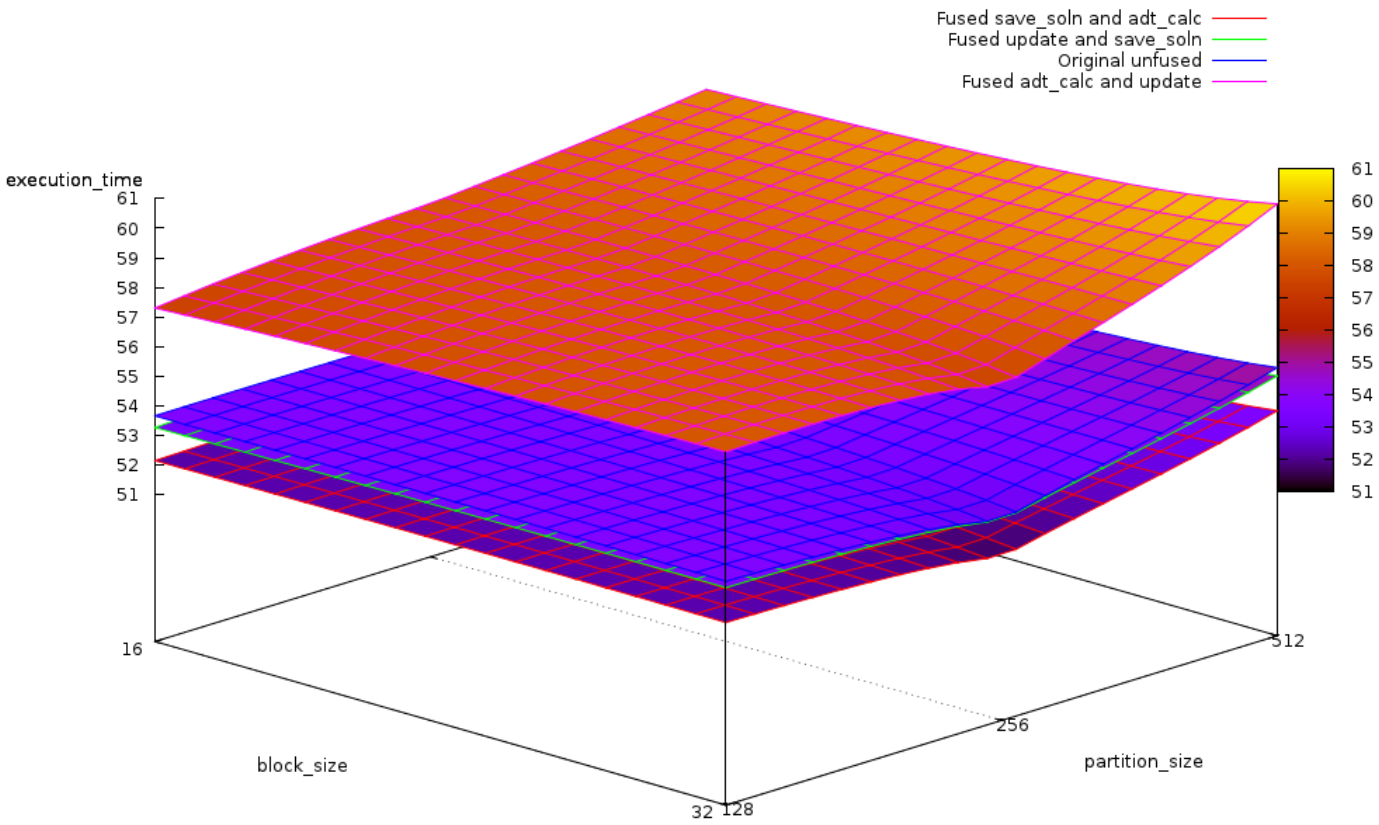


Figure 6.7: Xeon results that show the execution time of Airfoil with each fusion enabled separately, and the stock version when running with optimal block size and partition size pairs.

Figure 6.7 clearly shows an improvement in execution time of two of the fusions, compared to the stock version. However, here we observe that by fusing `update` and `adt_calc`, we actually decrease performance. As our initial

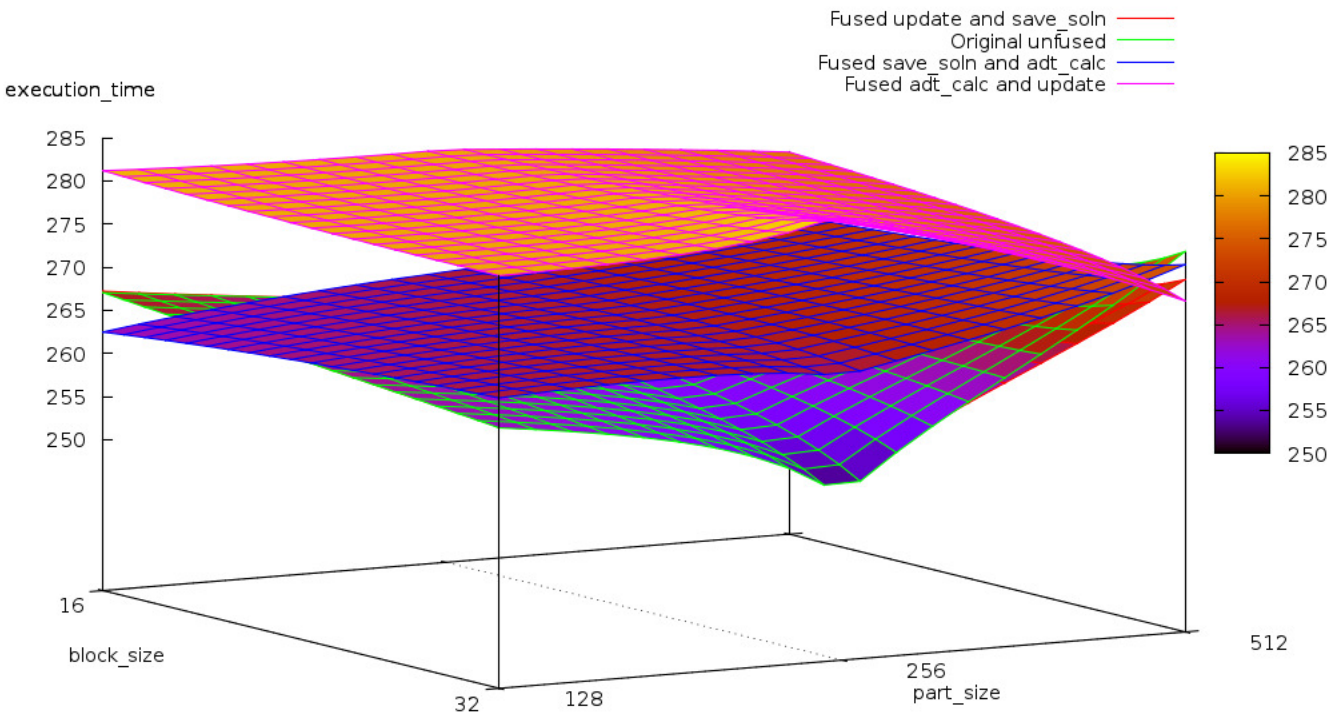


Figure 6.8: Core i7 results which show the execution time of Airfoil with fusions enabled individually, including the stock version when running with optimal block size and partition size pairs.

Airfoil version	cache-misses	branch-misses
Stock Airfoil	4 890 201 717	1 611 791 137
Fused save_soln and adt_calc	4 729 142 614	1 537 029 469
Fused adt_calc and update	4 251 236 541	1 563 408 439
Fused save_soln and update	5 017 608 868	1 569 090 758

Table 6.1: The initial analysis returned by Perf, containing only the number of cache and branch misses.

Airfoil version	instr. per cycle	L1-icache-load-misses	iTLB-load-misses
Stock Airfoil	1.02	1 197 267 639	18 902 629
Fused save_soln and adt_calc	1.03	1 251 404 877	18 481 245
Fused adt_calc and update	1.00	1 347 505 750	19 967 966
Fused save_soln and update	1.03	1 159 372 532	17 660 208

Table 6.2: The final analysis returned by Perf, which contains the number of instructions per cycle, instruction cache load and TLB load misses.

assumption was that all fusions are beneficial, we needed to understand why we observed different behaviour. We started by running the Airfoil versions on the Xeon with Perf [35]. We did not test the Core i7 results, as these were running in a virtual machine, which prevents us from successfully using profiling tools. Furthermore, those results exhibited significant variations in performance (within 36 runs of the same executable, with the same runtime parameters, we observed differences of up to 40 seconds or 14.3 % in execution time), thus we only include those mainly for reference. They were not used for fine tuning the machine learning and deterministic algorithms.

We initially looked for obvious factors that can decrease performance, such as cache misses and branch misses. However, as shown in Table 6.1, those did not provide us with much information, as the number of cache misses was higher than the one of the slow fusion, despite having an overall better execution time.

As a result, we looked further into the number of instructions executed, number of cycles, instruction cache load misses and instruction TLB load misses. The values in Table 6.2 gave us a clear indication that the issue was the complexity of the user supplied kernel.

We initially believed that all kernels are small enough to not cause bot-

Loop name	# of operations
save_soln	4
adt_calc	81
res_calc	96
bres_calc	88
update	13
Fused save_soln and adt_calc	85
Fused adt_calc and update	94
Fused save_soln and update	17

Table 6.3: The total number of operations in user-supplied kernel, for each loop.

tlenecks on CPUs, even after their fusion. However, these results clearly showed that we only have two beneficial fusions: `adt_calc` and `save_soln`, and `save_soln` and `update`. To confirm the results we obtained from Perf, we calculated the complexity of the user supplied kernel. We expressed it in terms of the number of operations performed: 1 operation for any mathematics function calls, operations and assignments, 2 for `if` statements and 3 for `for` loops. we present details as to how we chose these weights in Section 5.5.2: Weighting and complexity calculations. We got the following results 6.3.

Furthermore, as a final confirmation that not all user-supplied kernels are of appropriate size for CPUs, we checked the variations in performance and in instruction cache misses with respect to the stock Airfoil code. We observed that there is an approximately 4 % difference between the instruction cache misses variation and the average execution time variation. Excluding this difference, we saw that the 8.5 % variation of execution time translates accurately into the 12.5 % variation in icache misses. We show results in Table 6.4.

Airfoil version	L1-icache-load-misses	% variation	avg. exec time / s	% variation
stock Airfoil	1 197 267 639	0	52.7699	0
Fused save_soln and adt_calc	1 251 404 877	4.52	51.5258	−2.35
Fused adt_calc and update	1 347 505 750	12.54	57.3002	8.58
Fused save_soln and update	1 159 372 532	−3.16	52.7655	−0.008

Table 6.4: The correlation between the number of L1-icache-misses and execution time.

6.3 Tuning algorithm results

Once we confirmed the results and the correlation between user-supplied kernel size and variations in performance, we included this change in the weight calculations for both machine learning and deterministic algorithms. Therefore, our algorithms now produce optimal or close-to-optimal results for any given version of Airfoil. Due to our restricted data set, we can only guarantee the results for Airfoil, however, we strongly believe that any OP2 program can be run successfully through the algorithms and obtain optimal or close-to-optimal fusions and parameters. This is due to the nature of the OP2 programs, and the observed correlations between user kernel size and performance. We enforce this by confirming that the machine learning and, more importantly, the deterministic algorithm produce optimal or close-to-optimal results by generating appropriate fusions and runtime parameter values. We enforce the importance of the deterministic method, as it adjusts far better to previously unseen programs with different sets of loops. In Listing 6.3, we provide the results returned by our tuning script for each of the main versions of Airfoil. For completeness, we also show the associated complexities for each loop fusion candidate pair. Furthermore, no loop fusion with negative complexity is returned as feasible, even in the absence of any other fusions.

Listing 6.3: Here we present results returned by the tuning algorithm. As it can be seen, they are either optimal or close to optimal.

```
# Stock Airfoil
# no loops fusions, therefore no complexity
# solution
{'loops_to_fuse': [],
 'block_size': 32,
```

```
 'part_size': 256,
 'op_warpsize': 1}

# Loop unrolled Airfoil
# fusable loops: adt_calc and update, and save_soln and adt_calc
# solution
 {'loops_to_fuse': {'loop2': 'adt_calc',
                    'loop1': 'save_soln'},
   'block_size': 32,
   'part_size': 256,
   'op_warpsize': 1}

#complexity
[{'fusion': {'loop2': 'adt_calc', 'loop1': 'save_soln'},
   'complexity': 0.07193277310924273},
 {'fusion': {'loop2': 'adt_calc', 'loop1': 'update'},
   'complexity': -0.20612244897959275}]

# Loop unrolled Airfoil with code-block reordering
# fusable loops: adt_calc and save_soln, and adt_calc and update
# solution
{'loops_to_fuse': {'loop2': 'save_soln',
                   'loop1': 'update'},
 'block_size': 32,
 'part_size': 256,
 'op_warpsize': 1}


# complexity
[{'fusion': {'loop2': 'adt_calc', 'loop1': 'update'},
   'complexity': -0.20612244897959275},
 {'fusion': {'loop2': 'save_soln', 'loop1': 'update'},
   'complexity': 9.609778270509977}]
```

We do not require to add the script's results for the Airfoil versions which enable one optimization at a time, as we simulated the same behaviour by providing the initial Airfoil variations, which cover all these cases. Furthermore, we observe from the values of complexity that when having to choose between `adt_calc` and `save_soln`, and `update` and `save_soln`, the algorithm is always going to choose the latter. Despite having, in practice, a slightly worse performance, on paper the fusion appears to be better. This is due to the fact that they have 2 parameters in common, and a very low number of operations in the user supplied kernel. This can lead us to believe that there may not be a linear dependency between the user kernel's number of operations and the feasibility of the fusion.

The overall performance results for all versions of Airfoil, including those which have only one fusion enabled in the optimal case of `block_size =`

`32` and `partition_size = 256` are provided in Table 6.5. For completion purposes, we also include results from the Core i7 in Table 6.6.

Airfoil version	Execution time / s
stock Airfoil	52.7699
Fused save_soln and adt_calc, and adt_calc and update	55.6763
Fused save_soln and update, and adt_calc and update	57.9855
Fused save_soln and adt_calc	51.5258
Fused adt_calc and update	57.3002
Fused save_soln and update	52.7655

Table 6.5: The execution time for each variation of Airfoil we have tested with. We can clearly see that two of the fusions are more efficient than the stock version, thus giving us the desired overall performance improvement.

Airfoil version	Execution time / s
stock Airfoil	251.4098
Fused save_soln and adt_calc, and adt_calc and update	246.9143
Fused save_soln and update, and adt_calc and update	246.9143
Fused save_soln and adt_calc	265.4318
Fused adt_calc and update	284.2954
Fused save_soln and update	253.0470

Table 6.6: The Core i7 results of the execution time for each variation of Airfoil we have tested with. These results are included for reference only. We can observe that, despite each fusion being slower, individually, combined they give better execution times compared to the stock Airfoil. As this is highly unlikely, we attribute these differences to the high variations in runtime for each case, as observed earlier.

6.4 Complexity

We initially presented this approach as being more efficient than a brute-force one. A brute-force approach has exponential complexity, as we would be required to try every single possible value of every parameter with all others. The final algorithms we use, however, present a linear complexity with respect to the number of possible fusions or loops.

The machine learning algorithm initially parses all the known cases and generates its case-base. This is linear with respect to the number of cases. The similarity algorithm, has the same complexity, as it initially goes through all the cases it has stored and generates the weightings graphs. Secondly, it calculates the intersection of the known cases and the unsolved one. This intersection is once again, linear with respect to the number of cases. It also depends on the number of fusions, as we calculate the overall fusion complexity.

At the next step, we check the results by, again, calculating the complexities of all loop fusions, which is linear with respect to the number of possible fusions. The calculations for the runtime parameters are O(1).

The `retain` and `reuse` steps are both of O(1), therefore, our overall complexity is linear with respect to the number of possible loop fusions.

6.5 Summary

In this chapter we thoroughly tested the performance of multiple variations of Airfoil under a number of runtime parameters and highlighted the optimal results, which are returned by the machine learning and deterministic algorithms. We have therefore proved our hypothesis correct that, despite not having a significant performance improvement, by using machine learning and code analysis, we return in linear time optimal or close-to-optimal results which have better execution times than the stock Airfoil version. The next chapter presents a number of possible extensions and a case-study of a real-world application.

Chapter 7

Real-world applications - Hydra case-study

An important factor for any piece of work is its applicability onto real-world applications. Therefore, here we present the applicability of our tuning algorithm within the context of Hydra, based on work performed within Imperial College London's Software Performance Optimisation Group (SPO).

Hydra is an industrial application used by Rolls Royce, which is used to simulate inner turbomachinery components of jet engines. The application is highly complex and configurable, and therefore its computational complexity varies accordingly. We will focus our analysis on a standard computational fluid dynamics configuration scenario, as used by the SPO group.

Hydra contains 33 parallel loops in its main time-marching loop, which have a similar structure with the `op_par_loops` of OP2, thus making this an excellent discussion candidate for our application. Furthermore, the application was run on both GPUs and CPUs, by using CUDA and MPI, respectively. More specifically, this performance is benchmarked on two different high-end multicore nodes: dual 6-core Intel Xeon X5650 (Westmere) processors and an Intel Core i7 2600K (Sandy Bridge) processor. For evaluating the GPU performance an NVIDIA Tesla C2070 card is used, connected to a dual Intel Xeon X5650 host node. It has been observed that the baseline CUDA implementation is approximately 1.5× slower than the MPI solution running on 12 Westmere cores. After investigating the code, members of the SPO focused on a subset of 4 loops, which gave an overall poor performance. They soon realized that some of the main issues were related to complexity of user kernels associated with the loops. As the user kernels were too complex, the associated data would not fit in the shared memory, nor in the GPU's registers. As a result, register spillage occurred (once all 63 single precision/31 double precision registers have been used, data will be accessed directly from

global memory). A solution to this is to split each complex loop into multiple simpler ones.

At this point, we have the first possibility of using our algorithms for deciding which loops require splitting. Assuming our machine learning and deterministic algorithms would have knowledge of loop fission, we would be able to use the same weightings methods used by the loop fusion complexity calculation to determine whether loops should be split. More specifically, any loop with a negative complexity would have to be split. A correlation to Airfoil would be to consider a fused `adt_calc` and `update` as a given loop. We have determined that this fusion has a negative complexity, as outlined in Listing 6.3. As a result, if it were one initial loop we would decide to perform fission.

A second optimization that is related to our choice of optimizations is tuning the thread block size. The typical kernel consists of the following stages: staging in from device to shared memory, executing the user kernel and then staging back out from shared to device memory. The members of SPO group chose to increase the block size, as to provide a greater number of threads available for stage-in and stage-out phases, despite the fact that they might remain idle during user kernel execution. After setting the block size they chose to vary the partition size from 64 to 512, and obtained a performance speed-up of 1.93 over the baseline CUDA implementation.

This is the second point where our algorithms could produce good results. Despite the fact that we do not consider details of the stage-in and stage-out phases, nor, currently, the access descriptors of the loop's data parameters, we still obtain optimal performance on Airfoil. As a result, choosing to determine the block and partition size based on user kernel, and loop complexity could provide good results for comparison, and thus insight into the changes that we would need to make, if any, to our algorithms.

By considering the work done by the SPO group on Hydra, we have seen a greater scope for our project, and what changes we might need to introduce into our algorithms. These include low-level details of the architecture we are running our experiments on, and a more in-depth knowledge of the loop's parameters, such as the access descriptors.

Chapter 8

Conclusions

In this chapter we summarize the accomplishments of this project and present possible extensions. Finally, we reflect on the project and how it can guide future work in the area of optimizing compilers, especially with regards to the OP2 framework.

8.1 Achievements

Work on machine learning and deterministic algorithms to be used with the OP2 framework has been both rewarding and challenging. We started with a hypothesis that such methods can provide optimal or close-to-optimal results whilst only testing a small search space. The process of testing this has presented many challenges.

After choosing the, initially believed, very portable OpenCL as a back-end we encountered problems with the existing OP2 OpenCL runtime and handmade implementation, which returned incorrect results on CPUs. Once we fixed it, and updated the runtime to match OP2 standards, we started work on the untested compiler. This was also challenging, as the original generated code was not even compiling, let alone run. Nevertheless, we overcame this issue as well and we obtained a fully working OP2 to OpenCL compiler. These tools were necessary in order to allow us to automatically generate Airfoil variations.

At that stage, we started implementing a case-based reasoning system which would return us optimal runtime parameters and loop fusions. Halfway through developing the similarity measure, we observed a deterministic connection between a loops' parameters and user supplied kernel, and the feasibility of a fusion. We therefore implemented a machine learning algorithm that uses deterministic fail-safes in order to render optimal results. In order to aid

the machine learning algorithm, we created a small control-flow information analysis that would let the CBR system know which loops were fusable. We found little reason for attempting to integrate this in the machine learning algorithm, as it would overcomplicate the result and be error prone. As a result, we chose to initially decide which loops are fusable, and then the machine learning and deterministic algorithms decide whether the fusions are feasible.

Finally, we presented experimental results that back-up our initial hypothesis and confirm that there is a deterministic and machine learning connection between optimizations (both runtime and code transformations) and the code structure. Furthermore, we show that the runtime parameters must take in consideration any code transformation optimizations that are performed.

As an extension, we presented a case-study of the applicability of our approach onto real-world applications, such as Hydra.

In summary, we make the following contributions:

- We adjusted the existing OP2 OpenCL runtime so that it successfully runs on both CPUs and GPUs, and we structured it according to OP2 standards.
- We fixed the untested, broken OP2 OpenCL compiler.
- We showed that a machine learning and deterministic approach to code analysis can successfully provide performance improvements.
- We explored the performance gains on variations of Airfoil.
- We discussed the applicability of our results on Hydra.

8.2 Further work

This project is only a proof of concept for the use of machine learning and deterministic algorithms to get speed-ups. As a result, there are many possible extensions for this project. The main ones are the following:

- As we have observed whilst designing the deterministic algorithm, we needed to experimentally find values for the weightings. A good extension would be to use a machine learning algorithm at this stage to derive the weightings. This might prove more reliable across various programs, and it might derive better overall results, as it would not be bound to experimental data.

- This project would greatly benefit by having more optimizations added to it, such as loop fission, automatic loop unrolling and code-block reordering. This would allow us to get overall better results, as we could weight optimizations against each other and choose the very best from a pool of choices. Furthermore, adding extra optimizations which are enabled by loop fusion could increase the benefits obtained by fusion, and could potentially make currently unfeasible fusions perform better than stock versions of the loops.

- As we saw in the Evaluation section, the characteristics of the machine we are running our experiments on matter. As a result, it would be beneficial to add this extra low level information into the machine learning and deterministic algorithms. More knowledge of the underlying architecture should allow us to better derive the runtime parameters and feasible optimizations. Furthermore, a further analysis of the loops' parameters could give us important information with respect to the complexity of the generated kernels.

- Another good extension would be to add support for multiple back-ends and front-ends. At the moment, all the analysis we are performing is on OP2 C++ code. If we performed experiments by using CUDA, OpenMP and other available back-ends, we would be able to provide extra support. Furthermore, adding OP2 Fortran support merely involves adapting the initial parsing of the file. As a result, we would be able to generate better results, which can take in consideration performance specifics of each back-end.

8.3 Final remarks

We hope the proof of concept of using machine learning and deterministic approaches to determine optimal and close-to-optimal optimizations in linear time has provided a starting ground for future work into optimizing compilers. As our evaluation shows, such methods can prove successful and significantly decrease the optimization search space. This work can be directly and successfully applied to the existing OP2 framework, thus providing a novel extension to it. Furthermore, this methodology is easily extensible to other architectures and front and back-ends, thus provides an excellent starting point for extensions, which can be used to further improve OP2's performance.

Chapter 9

Bibliography

[1] Geoff Koch. Transitioning to Multi-Core Chip Architecture. http://software.intel.com/en-us/articles/transitioning-to-multi-core-chip-architecture/?wapkw=(multi-core+architecture). accessed June 19, 2012.

[2] Nvidia Corporation. GPU Computing. http://www.nvidia.com/object/GPU_Computing.html. accessed June 19, 2012.

[3] Apple Inc. and The Khronos Group Inc. OpenCL. http://www.khronos.org/opencl/. accessed June 19, 2012.

[4] Mike Giles. OP2. http://people.maths.ox.ac.uk/gilesm/op2/. accessed June 19, 2012.

[5] M.B. Giles, D. Ghate, and M.C. Duta. Using automatic differentiation for adjoint CFD code development. *computational Fluid Dynamics Journal*, (16):434–443, 2008.

[6] Andrew Binstock. Multi-Core Processor Architecture Explained. http://software.intel.com/en-us/articles/multi-core-processor-architecture-explained/?wapkw=(multi-core+architecture). accessed June 19, 2012.

[7] Chris Lomont. Introduction to Intel Advanced Vector Extensions. http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/. accessed June 19, 2012.

[8] Sam Siewert. Using Intel©Streaming SIMD Extensions and Intel©Integrated Performance Primitives to Accelerate Algorithms. http://software.intel.com/en-us/

articles/using-intel-streaming-simd-extensions-and\
-intel-integrated-performance-primitives-to-accelerate-algorithms/.
accessed June 19, 2012.

[9] Nvidia Corporation. CUDA Zone. http://developer.nvidia.com/category/zone/cuda-zone. accessed June 19, 2012.

[10] Inc. Advanced Micro Devices. AMD Developer Zone. http://developer.amd.com/zones/Pages/default.aspx. accessed June 19, 2012.

[11] Nvidia Corporation. CUDA. http://www.nvidia.com/object/cuda_home_new.html. accessed June 19, 2012.

[12] Nvidia Corporation. CUDA Description. http://developer.nvidia.com/what-cuda. accessed June 19, 2012.

[13] Microsoft Corporation. DirectCompute. http://developer.nvidia.com/directcompute. accessed June 19, 2012.

[14] Advanced Micro Devices, Inc. OpenCL Introduction. http://www.amd.com/uk/products/technologies/stream-technology/opencl/Pages/opencl-intro.aspx. accessed June 19, 2012.

[15] Carlo Bertolli, Adam Betts, Gihan Mudalige, Mike Giles, and Paul Kelly. Design and Performance of the OP2 Library for Unstructured Mesh Applications.

[16] M.B. Giles, G.R. Mudalige, Z. Sharif, G. Markall, and P.H.J Kelly. Performance Analysis and Optimisation of the OP2 Framework on Many-core Architectures. *The Computer Journal*, 2010.

[17] OpenMP Architecture Review Board. OpenMP. http://openmp.org/wp/. accessed June 19, 2012.

[18] Stanford University. Liszt: A DSL for solving mesh-based PDEs. http://liszt.stanford.edu/. accessed June 19, 2012.

[19] Martin Odersky and Programming Methods Laboratory of EPFL. Scala Programming Language. http://www.scala-lang.org/. accessed June 19, 2012.

[20] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Alan LaMielle. Introducing the Sparse Polyhedral Framework (SPF). Front Range Architecture Compilers Tools and Languages Fall 2009, December 5, 2009.

[21] Michelle Mills Strout. Automating Run-Time Reordering Transformations with the Sparse Polyhedral Framework (SPF) and Arbitrary Task Graphs. Imperial College London, November 21, 2011.

[22] Ravi Mirchandaney, Joel H. Saltz, Roger M. Smith, David M. Nicol, and Kay Crowley. Principles of Runtime Support for Parallel Processors. 1988.

[23] Sajal K. Das and Aisheng Mao. A Theoretical Network Model and the Hamming Cube Networks. pages 18–22. Parallel Processing Symposium, 1994. Proceedings., Eighth International, April 26-29, 1994.

[24] Jonathan Roelofs. IEGenCC and IEGen. `http://www.cs.colostate.edu/~roelofs/iegencc_writeup.php`. accessed June 19, 2012.

[25] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Rescheduling for Locality in Sparse Matrix Computations. The 2001 International Conference on Computational Science, May 28-30, 2001.

[26] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. Sparse Tiling For Stationary Iterative Methods. *The International Journal on High Performance Computing Applications*, 18(1):95–113, 2004.

[27] Cedric Bastoul. Cloog. Code Generation in the Polyhedral Model Is Easier Than You Think, PACT '04 Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques `www.CLooG.org`, 2004. accessed June 19, 2012.

[28] Tom Mitchell. *Machine Learning.* McGraw-Hill Science/Engineering/-Math, 1st edition, 1997.

[29] R.C. Schank. Dynamic memory: A theory of reminding and learning in coputers and people. *Cambridge University Press*, 1982.

[30] R.C. Schank. Memory-based expert systems. *Technical Report (AFOSR. TR. 84-0814), Yale University, New Haven, USA*, 1984.

[31] Maja Pantic. Machine Learning Course - Inductive Logic Programming. `http://ibug.doc.ic.ac.uk/media/uploads/documents/courses/ml-lecture4.pdf`. accessed June 19, 2012.

[32] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F.P. O'Boyle. Portable Compiler Optimisation Across

Embedded Programs and Microarchitectures using Machine Learning. *MICRO'09*, 2009.

[33] Ben Spencer. Flamingo Auto-Tuning. `http://mistymountain.co.uk/flamingo/`. accessed June 19, 2012.

[34] Dan Quinlan, Chunhua Liao, Justin Tao, Thomas Panas, Jeremiah Willcock, Markus Schordan, Qing Yi, and Rich Vuduc. Rose. `http://www.rosecompiler.org/`. accessed June 19, 2012.

[35] Perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`. accessed June 19, 2012.

Appendix A

Code samples

A.1 OP2 Airfoil

Here we show the full OP2 Airfoil C++ code, with the computational kernels.

Listing A.1: OP2 Airfoil C++ code

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <math.h>
5
6  // global constants
7
8  float gam, gm1, cfl, eps, mach, alpha, qinf[4];
9
10 //
11 // OP header file
12 //
13
14 #include "op_lib_cpp.h"
15 #include "op_seq.h"
16
17 //
18 // kernel routines for parallel loops
19 //
20
21 #include "save_soln.h"
22 #include "adt_calc.h"
23 #include "res_calc.h"
24 #include "bres_calc.h"
25 #include "update.h"
26
27 // main program
28
```

```
29 int main(int argc, char **argv)
30 {
31   // OP initialisation
32   op_init(argc,argv,2);
33
34   int    *becell, *ecell,  *bound, *bedge, *edge, *cell;
35   float  *x, *q, *qold, *adt, *res;
36
37   int    nnode,ncell,nedge,nbedge,niter;
38   float  rms;
39
40   //timer
41   double cpu_t1, cpu_t2, wall_t1, wall_t2;
42
43   // read in grid
44   op_printf("reading in grid \n");
45
46   FILE *fp;
47   if ( (fp = fopen("./new_grid.dat","r")) == NULL) {
48     op_printf("can't open file new_grid.dat\n"); exit(-1);
49   }
50
51   if (fscanf(fp,"%d %d %d %d \n",&nnode, &ncell, &nedge, &nbedge) != 4) {
52     op_printf("error reading from new_grid.dat\n"); exit(-1);
53   }
54
55   cell   = (int *) malloc(4*ncell*sizeof(int));
56   edge   = (int *) malloc(2*nedge*sizeof(int));
57   ecell  = (int *) malloc(2*nedge*sizeof(int));
58   bedge  = (int *) malloc(2*nbedge*sizeof(int));
59   becell = (int *) malloc(  nbedge*sizeof(int));
60   bound  = (int *) malloc(  nbedge*sizeof(int));
61
62   x      = (float *) malloc(2*nnode*sizeof(float));
63   q      = (float *) malloc(4*ncell*sizeof(float));
64   qold   = (float *) malloc(4*ncell*sizeof(float));
65   res    = (float *) malloc(4*ncell*sizeof(float));
66   adt    = (float *) malloc(  ncell*sizeof(float));
67
68   for (int n=0; n<nnode; n++) {
69     if (fscanf(fp,"%f %f \n",&x[2*n], &x[2*n+1]) != 2) {
70       op_printf("error reading from new_grid.dat\n"); exit(-1);
71     }
72   }
73
74   for (int n=0; n<ncell; n++) {
75     if (fscanf(fp,"%d %d %d %d \n",&cell[4*n  ], &cell[4*n+1],
76                                    &cell[4*n+2], &cell[4*n+3]) != 4) {
77       op_printf("error reading from new_grid.dat\n"); exit(-1);
```

```
78      }
79    }
80
81    for (int n=0; n<nedge; n++) {
82      if (fscanf(fp,"%d %d %d %d \n",&edge[2*n], &edge[2*n+1],
83                                     &ecell[2*n],&ecell[2*n+1]) != 4) {
84        op_printf("error reading from new_grid.dat\n"); exit(-1);
85      }
86    }
87
88    for (int n=0; n<nbedge; n++) {
89      if (fscanf(fp,"%d %d %d %d \n",&bedge[2*n],&bedge[2*n+1],
90                                     &becell[n], &bound[n]) != 4) {
91        op_printf("error reading from new_grid.dat\n"); exit(-1);
92      }
93    }
94
95    fclose(fp);
96
97    // set constants and initialise flow field and residual
98
99    op_printf("initialising flow field \n");
100
101   gam = 1.4f;
102   gm1 = gam - 1.0f;
103   cfl = 0.9f;
104   eps = 0.05f;
105
106   float mach  = 0.4f;
107   float alpha = 3.0f*atanf(1.0f)/45.0f;
108   float p     = 1.0f;
109   float r     = 1.0f;
110   float u     = sqrtf(gam*p/r)*mach;
111   float e     = p/(r*gm1) + 0.5f*u*u;
112
113   qinf[0] = r;
114   qinf[1] = r*u;
115   qinf[2] = 0.0f;
116   qinf[3] = r*e;
117
118   for (int n=0; n<ncell; n++) {
119     for (int m=0; m<4; m++) {
120       q[4*n+m] = qinf[m];
121     res[4*n+m] = 0.0f;
122     }
123   }
124
125   // declare sets, pointers, datasets and global constants
126
```

```
127   op_set nodes  = op_decl_set(nnode,  "nodes");
128   op_set edges  = op_decl_set(nedge,  "edges");
129   op_set bedges = op_decl_set(nbedge, "bedges");
130   op_set cells  = op_decl_set(ncell,  "cells");
131
132   op_map pedge   = op_decl_map(edges, nodes,2,edge,  "pedge");
133   op_map pecell  = op_decl_map(edges, cells,2,ecell, "pecell");
134   op_map pbedge  = op_decl_map(bedges,nodes,2,bedge, "pbedge");
135   op_map pbecell = op_decl_map(bedges,cells,1,becell,"pbecell");
136   op_map pcell   = op_decl_map(cells, nodes,4,cell,  "pcell");
137
138   op_dat p_bound = op_decl_dat(bedges,1,"int"  ,bound,"p_bound");
139   op_dat p_x     = op_decl_dat(nodes ,2,"float",x    ,"p_x");
140   op_dat p_q     = op_decl_dat(cells ,4,"float",q    ,"p_q");
141   op_dat p_qold  = op_decl_dat(cells ,4,"float",qold ,"p_qold");
142   op_dat p_adt   = op_decl_dat(cells ,1,"float",adt  ,"p_adt");
143   op_dat p_res   = op_decl_dat(cells ,4,"float",res  ,"p_res");
144
145   op_decl_const(1,"float",&gam  );
146   op_decl_const(1,"float",&gm1  );
147   op_decl_const(1,"float",&cfl  );
148   op_decl_const(1,"float",&eps  );
149   op_decl_const(1,"float",&mach );
150   op_decl_const(1,"float",&alpha);
151   op_decl_const(4,"float",qinf  );
152
153   op_diagnostic_output();
154
155   //initialise timers for total execution wall time
156   op_timers(&cpu_t1, &wall_t1);
157
158   // main time-marching loop
159
160   niter = 1000;
161
162   for(int iter=1; iter<=niter; iter++) {
163
164     // save old flow solution
165
166     op_par_loop(save_soln,"save_soln", cells,
167       op_arg_dat(p_q,   -1,OP_ID, 4,"float",OP_READ ),
168       op_arg_dat(p_qold,-1,OP_ID, 4,"float",OP_WRITE));
169
170     // predictor/corrector update loop
171
172     for(int k=0; k<2; k++) {
173
174       // calculate area/timstep
175
```

```
176         op_par_loop(adt_calc,"adt_calc",cells,
177             op_arg_dat(p_x,   0,pcell, 2,"float",OP_READ ),
178             op_arg_dat(p_x,   1,pcell, 2,"float",OP_READ ),
179             op_arg_dat(p_x,   2,pcell, 2,"float",OP_READ ),
180             op_arg_dat(p_x,   3,pcell, 2,"float",OP_READ ),
181             op_arg_dat(p_q,  -1,OP_ID, 4,"float",OP_READ ),
182             op_arg_dat(p_adt,-1,OP_ID, 1,"float",OP_WRITE));
183
184         // calculate flux residual
185
186         op_par_loop(res_calc,"res_calc",edges,
187             op_arg_dat(p_x,    0,pedge, 2,"float",OP_READ),
188             op_arg_dat(p_x,    1,pedge, 2,"float",OP_READ),
189             op_arg_dat(p_q,    0,pecell,4,"float",OP_READ),
190             op_arg_dat(p_q,    1,pecell,4,"float",OP_READ),
191             op_arg_dat(p_adt,  0,pecell,1,"float",OP_READ),
192             op_arg_dat(p_adt,  1,pecell,1,"float",OP_READ),
193             op_arg_dat(p_res,  0,pecell,4,"float",OP_INC ),
194             op_arg_dat(p_res,  1,pecell,4,"float",OP_INC ));
195
196         op_par_loop(bres_calc,"bres_calc",bedges,
197             op_arg_dat(p_x,     0,pbedge, 2,"float",OP_READ),
198             op_arg_dat(p_x,     1,pbedge, 2,"float",OP_READ),
199             op_arg_dat(p_q,     0,pbecell,4,"float",OP_READ),
200             op_arg_dat(p_adt,   0,pbecell,1,"float",OP_READ),
201             op_arg_dat(p_res,   0,pbecell,4,"float",OP_INC ),
202             op_arg_dat(p_bound,-1,OP_ID  ,1,"int",  OP_READ));
203
204         // update flow field
205
206         rms = 0.0;
207
208         op_par_loop(update,"update",cells,
209             op_arg_dat(p_qold,-1,OP_ID, 4,"float",OP_READ ),
210             op_arg_dat(p_q,   -1,OP_ID, 4,"float",OP_WRITE),
211             op_arg_dat(p_res, -1,OP_ID, 4,"float",OP_RW   ),
212             op_arg_dat(p_adt, -1,OP_ID, 1,"float",OP_READ ),
213             op_arg_gbl(&rms,1,"float",OP_INC));
214       }
215
216       // print iteration history
217       rms = sqrtf(rms/(float) op_get_size(cells));
218       if (iter%100 == 0)
219         op_printf(" %d  %10.5e \n",iter,rms);
220     }
221
222     op_timers(&cpu_t2, &wall_t2);
223     op_timing_output();
224     op_printf("Max total runtime = \n%f\n",wall_t2-wall_t1);
```

```
225
226   op_exit();
227 }
```

Listing A.2: C++ save_soln kernel

```
1  inline void save_soln(float *q, float *qold){
2    for (int n=0; n<4; n++) qold[n] = q[n];
3  }
```

Listing A.3: C++ adt_calc kernel

```
1  inline void adt_calc(float *x1,float *x2,float *x3,float *x4,
2    float *q,float *adt){
3    float dx,dy, ri,u,v,c;
4
5    ri =  1.0f/q[0];
6    u  =   ri*q[1];
7    v  =   ri*q[2];
8    c  = sqrtf(gam*gm1*(ri*q[3]-0.5f*(u*u+v*v)));
9
10   dx = x2[0] - x1[0];
11   dy = x2[1] - x1[1];
12   *adt  = fabsf(u*dy-v*dx) + c*sqrtf(dx*dx+dy*dy);
13
14   dx = x3[0] - x2[0];
15   dy = x3[1] - x2[1];
16   *adt += fabsf(u*dy-v*dx) + c*sqrtf(dx*dx+dy*dy);
17
18   dx = x4[0] - x3[0];
19   dy = x4[1] - x3[1];
20   *adt += fabsf(u*dy-v*dx) + c*sqrtf(dx*dx+dy*dy);
21
22   dx = x1[0] - x4[0];
23   dy = x1[1] - x4[1];
24   *adt += fabsf(u*dy-v*dx) + c*sqrtf(dx*dx+dy*dy);
25
26   *adt = (*adt) / cfl;
27 }
```

Listing A.4: C++ res_calc kernel

```
1  inline void res_calc(float *x1,  float *x2,  float *q1,  float *q2,
2    float *adt1,float *adt2,float *res1,float *res2) {
3    float dx,dy,mu, ri, p1,vol1, p2,vol2, f;
4
5    dx = x1[0] - x2[0];
6    dy = x1[1] - x2[1];
7
8    ri   = 1.0f/q1[0];
```

```
  p1    = gm1*(q1[3]-0.5f*ri*(q1[1]*q1[1]+q1[2]*q1[2]));
  vol1 =  ri*(q1[1]*dy - q1[2]*dx);

  ri    = 1.0f/q2[0];
  p2    = gm1*(q2[3]-0.5f*ri*(q2[1]*q2[1]+q2[2]*q2[2]));
  vol2 =  ri*(q2[1]*dy - q2[2]*dx);

  mu = 0.5f*((*adt1)+(*adt2))*eps;

  f = 0.5f*(vol1* q1[0] + vol2* q2[0]) + mu*(q1[0]-q2[0]);
  res1[0] += f;
  res2[0] -= f;
  f = 0.5f*(vol1* q1[1] + p1*dy + vol2* q2[1] + p2*dy) + mu*(q1[1]-q2[1]);
  res1[1] += f;
  res2[1] -= f;
  f = 0.5f*(vol1* q1[2] - p1*dx + vol2* q2[2] - p2*dx) + mu*(q1[2]-q2[2]);
  res1[2] += f;
  res2[2] -= f;
  f = 0.5f*(vol1*(q1[3]+p1) + vol2*(q2[3]+p2)) + mu*(q1[3]-q2[3]);
  res1[3] += f;
  res2[3] -= f;
}
```

Listing A.5: C++ bres_calc kernel

```
inline void bres_calc(float *x1,  float *x2,  float *q1,
  float *adt1,float *res1,int *bound) {
  float dx,dy,mu, ri, p1,vol1, p2,vol2, f;

  dx = x1[0] - x2[0];
  dy = x1[1] - x2[1];

  ri = 1.0f/q1[0];
  p1 = gm1*(q1[3]-0.5f*ri*(q1[1]*q1[1]+q1[2]*q1[2]));

  if (*bound==1) {
    res1[1] += + p1*dy;
    res1[2] += - p1*dx;
  }
  else {
    vol1 =  ri*(q1[1]*dy - q1[2]*dx);

    ri    = 1.0f/qinf[0];
    p2    = gm1*(qinf[3]-0.5f*ri*(qinf[1]*qinf[1]+qinf[2]*qinf[2]));
    vol2 =  ri*(qinf[1]*dy - qinf[2]*dx);

    mu = (*adt1)*eps;

    f = 0.5f*(vol1* q1[0] + vol2* qinf[0]) + mu*(q1[0]-qinf[0]);
    res1[0] += f;
```

```
26      f = 0.5f*(vol1* q1[1] + p1*dy + vol2* qinf[1] + p2*dy)
27        + mu*(q1[1]-qinf[1]);
28      res1[1] += f;
29      f = 0.5f*(vol1* q1[2] - p1*dx + vol2* qinf[2] - p2*dx)
30        + mu*(q1[2]-qinf[2]);
31      res1[2] += f;
32      f = 0.5f*(vol1*(q1[3]+p1) + vol2*(qinf[3]+p2)) + mu*(q1[3]-qinf[3]);
33      res1[3] += f;
34    }
35  }
```

Listing A.6: C++ update kernel

```
1  inline void update(float *qold, float *q, float *res,
2    float *adt, float *rms){
3    float del, adti;
4
5    adti = 1.0f/(*adt);
6
7    for (int n=0; n<4; n++) {
8      del    = adti*res[n];
9      q[n]   = qold[n] - del;
10     res[n] = 0.0f;
11     *rms  += del*del;
12   }
13 }
```

A.2 OP2 Tuner Runtime Support

Listing A.7: This is a header file we created for the OP2 Tuner runtime support, `op_lib_tuner.h`.

```
1
2  /*
3   * The core of the runtime op_tuner
4   */
5
6  typedef enum {ANY, CPU, GPU, ACCELERATOR} arch;
7
8  typedef struct {
9    int   block_size;
10   int   part_size;
11   int   cache_line_size;
12   int   op_warpsize;
13   arch  architecture;
14   char const *name;
15   int   loop_tuner;
```

```
16    int    active;
17  } op_tuner;
18
19  struct node {
20    op_tuner *OP_tuner;
21    struct node * next;
22  };
23
24  /*
25   * method declarations necessary for the op_tuner.
26   * variables necessary for the tuners.
27   * Also, we are externalizing OP_cache_line_size as it can be
28   * manipulated by the runtime op_tuner.
29   */
30
31  extern int OP_cache_line_size;
32  //extern node *head, *current;
33  //extern op_tuner* OP_global_tuner;
34
35
36  #ifdef __cplusplus
37  extern "C" {
38  #endif
39
40  op_tuner * op_tuner_core(char const *);
41
42  op_tuner * op_tuner_get(char const *);
43
44  op_tuner * op_create_tuner(char const *);
45
46  op_tuner * op_create_global_tuner();
47
48  op_tuner * op_get_global_tuner();
49  #ifdef __cplusplus
50  }
51  #endif
52
53  #endif
```

A.3 Split Files Script

This is the core of the Python script which splits the generated host and kernel code file into two appropriate files.

Listing A.8: This is the core of a Python script which splits the compiler generated host and kernel code file into two appropriate files.

```
1
2  #!/usr/bin/env python
```

```
3
4  from os import sep
5  from sys import path, argv
6  import sys
7  import string
8
9  gen_file = open("rose_opencl_code_opencl.cpp", 'r');
10 gen_lines = gen_file.readlines();
11
12 lines_host_file = [];
13 lines_kernel_file = [];
14
15 ....
16
17 def addLinesToFileAndRemoveFromList(file_lines, gen_lines):
18   openCurlyBraces = 0;
19   closedCurlyBraces = 0;
20   line_index = gen_lines.index(line);
21   initial_line = line_index;
22   while not isBeginningOfStatement(gen_lines[line_index]):
23     line_index += 1;
24   openCurlyBraces = 1;
25   if isEndOfStatement(gen_lines[line_index]):
26     closedCurlyBraces += 1;
27   line_index += 1;
28   while openCurlyBraces != closedCurlyBraces:
29     if isBeginningOfStatement(gen_lines[line_index]):
30       openCurlyBraces += 1;
31     if isEndOfStatement(gen_lines[line_index]):
32       closedCurlyBraces += 1;
33     line_index += 1;
34   final_line = line_index;
35   while final_line-initial_line > 0:
36     if not isRogueLine(gen_lines[initial_line]):
37       file_lines.append(gen_lines[initial_line]);
38     gen_lines.remove(gen_lines[initial_line]);
39     final_line -= 1;
40   return [file_lines, gen_lines];
41
42 ...
43
44 openCurlyBraces = 0;
45 closedCurlyBraces = 0;
46
47 nestingLevel = 1;
48 index = 0;
49
50 for line in gen_lines:
51   if isIfDefStatement(line):
```

```
52      initial_line = gen_lines.index(line);
53      line_index = initial_line;
54      while not isEndIfDefStatement(gen_lines[line_index]):
55        line_index += 1;
56      while line_index - initial_line > 0:
57        lines_host_file.append(gen_lines[initial_line]);
58        gen_lines.remove(gen_lines[initial_line]);
59        line_index -= 1;
60      lines_host_file.append(gen_lines[initial_line]);
61    if isHashDefStatement(line):
62      if string.find(line, 'ROUND_UP') != -1 or
63         string.find(line, 'MIN') != -1 or
64         string.find(line, 'ZERO_float') != -1:
65        lines_kernel_file.append(line);
66    if isIncludeStatement(line):
67      lines_host_file.append(line);
68    if isExternStatement(line):
69      lines_host_file.append(line);
70    if isDeclarationStatement(line):
71      lines_host_file.append(line);
72    if isInlineVoidMethod(line):
73      [lines_kernel_file, gen_lines] =
74        addLinesToFileAndRemoveFromList(lines_kernel_file, gen_lines);
75    if isKernelMethod(line):
76      [lines_kernel_file, gen_lines] =
77        addLinesToFileAndRemoveFromList(lines_kernel_file, gen_lines);
78    if isHostMethod(line):
79      [lines_host_file, gen_lines] =
80        addLinesToFileAndRemoveFromList(lines_host_file, gen_lines);
81
82  hosts_file = open('rose_opencl_hosts.cpp','w');
83  kernels_file = open('rose_opencl_code_opencl.cl','w');
84
85  hosts_file.writelines(lines_host_file);
86
87  kernels_file.writelines(lines_kernel_file);
```

A.4 Tuner script

Here you can find a comprehensive copy of the tuner script, that contains all the weighting algorithms, machine learning and deterministic approaches, control flow information generation and loop fusion evaluation. We only included the core, relevant parts of the script.

Listing A.9: This is the core of the Python tuner script which contains all the analysis we perform: from OP2 text parsing, control flow information

and loop fusion analysis, to machine learning and deterministic optimization evaluations. We only show core methods of the control flow analysis, machine learning and deterministic algorithms.

```
...
class Arch:
  ANY = 0
  CPU = 1
  GPU = 2
  ACCELERATOR = 3

...
# now we need to decide if we have any fusable loops
# we perform basic control flow

...
def addToCFInfo(statementName, statementDepth, CFInfoRootNode):
  currentDepth = 1;
  currentNode = CFInfoRootNode;
  while (currentDepth < statementDepth and
     len(currentNode['children']) > 0):
    currentNode =
        currentNode['children'][len(currentNode['children'])-1];
    currentDepth = currentDepth + 1;

  CFInfoNode = {
                  'name' : statementName,
                  'children' : [],
                  'depth' : statementDepth
              }
  currentNode['children'].append(CFInfoNode);
  return CFInfoRootNode;

def getCFInfoNode(nodeName, CFInfoRootNode):
  currentNode = CFInfoRootNode;
  if currentNode['name'] == nodeName:
    return currentNode;
  else:
    desiredNode = None;
    for node in currentNode['children']:
      desiredNode = getCFInfoNode(nodeName, node);
      if desiredNode != None and desiredNode['name'] == nodeName:
        return desiredNode;

def getParentCFInfoNode(nodeName, CFInfoRootNode):
  currentNode = CFInfoRootNode;
  for node in currentNode['children']:
    if node['name'] == nodeName:
      return currentNode;
```

```
46    desiredNode = None;
47    for node in currentNode['children']:
48      desiredNode = getParentCFInfoNode(nodeName, node);
49      if desiredNode != None:
50        for node in desiredNode['children']:
51          if node['name'] == nodeName:
52            return desiredNode;
53
54  ...
55
56  #CFInfo
57  CFInfoRootNode = {  'name' : 'ROOT',
58                      'children' : [],
59                      'depth' : 0
60                   }
61  nestingLevel = 1;
62  orderOfAddition = [];
63  opParLoopNameMap = [];
64  index = 0;
65  indexOpParLoops = 0;
66  for line in core_lines:
67    if isForLoopStatement(line):
68      new_name = "for" + str(index);
69      index = index + 1;
70      CFInfoRootNode =
71          addToCFInfo(new_name, nestingLevel, CFInfoRootNode);
72      nestingLevel = nestingLevel + 1;
73      orderOfAddition.append(new_name);
74    else:
75      if isOpParLoopStatement(line):
76        line_comp = line.split('(');
77        line_comp = line_comp[1];
78        line_comp = line_comp.split(',');
79        new_name='';
80        for comp in line_comp:
81          if '"' in comp:
82            comp = comp.strip('"');
83            new_name = comp;
84        opParLoopNameMap.append({'original' : new_name,
85          'tag' : new_name + str(indexOpParLoops)});
86        new_name += str(indexOpParLoops);
87        CFInfoRootNode =
88            addToCFInfo(new_name, nestingLevel, CFInfoRootNode);
89        orderOfAddition.append(new_name);
90        indexOpParLoops += 1;
91      else:
92        if isEndOfStatement(line) and
93           len(CFInfoRootNode['children']) > 0:
94          nestingLevel = nestingLevel - 1;
```

```
95
96  ...
97  # now we have the Control Flow Information so
98  # we can decide which loops we can fuse
99  ...
100 fusable_pairs = [];
101
102 for index in range(len(loops_in_order)-1):
103   loop1 = getCFInfoNode(loops_in_order[index], CFInfoRootNode);
104   depthLoop1 = loop1['depth'];
105   loop2 = getCFInfoNode(loops_in_order[index+1], CFInfoRootNode);
106   depthLoop2 = loop2['depth'];
107   if depthLoop1 == depthLoop2:
108     # need to check if there are any other statements in between
109     parentNode =
110         getParentCFInfoNode(loops_in_order[index], CFInfoRootNode);
111     childrenNodes = parentNode['children'];
112     child1Index = childrenNodes.index(loop1);
113     child2Index = childrenNodes.index(loop2);
114     if child1Index + 1 == child2Index:
115       #pair's fusable - add to fusable_pairs list
116       fusable_pair = {
117                       'loop1' : loop1['name'],
118                       'loop2' : loop2['name']
119                      }
120       fusable_pairs.append(fusable_pair);
121
122 ...
123
124 # we now do parameter analysis to see if we can/should really fuse them
125 # define CBR
126
127 CBRSystem = []
128
129 # case base:
130
131 trainingCases = [CBRCase1, CBRCase2, CBRCase1, CBRCase3, CBRCase4];
132 results = [CBRSolution1, CBRSolution2, CBRSolution1,
133    CBRSolution3, CBRSolution4];
134
135 def retrieveTrainingData():
136   return [trainingCases, results];
137
138 # init CBR
139
140 ...
141
142 def fusableLoopsWeighting(CBRCase):
143   weighting = 0;
```

```
144   for pair in CBRCase['fusable_pairs']:
145     loop1 = getLoopInfo(pair['loop1'], CBRCase['op_par_loops']);
146     loop2 = getLoopInfo(pair['loop2'], CBRCase['op_par_loops']);
147     if opParLoopArrayMatch(loop1, loop2):
148       weighting += 1;
149   return weighting;
150 
151 ...
152 
153 def calculateLoopFusionComplexity(loop1, loop2, arch):
154   complexity = 0;
155   noOfEqualArgs = 0;
156   hasEqual = [0 for i in range (len(loop2['op_arg_dat']))];
157   for dataArg1 in loop1['op_arg_dat']:
158     for dataArg2 in loop2['op_arg_dat']:
159       if opArgDatMatch(dataArg1, dataArg2):
160         noOfEqualArgs += 1;
161         hasEqual[loop2['op_arg_dat'].index(dataArg2)] +=  1;
162   totalNoOfArgs = len(loop1['op_arg_dat']) +
163      len(loop2['op_arg_dat']);
164   propEqualArgsLoop1 = noOfEqualArgs/len(loop1['op_arg_dat']);
165   propEqualArgsLoop2 = noOfEqualArgs/len(loop2['op_arg_dat']);
166   matched = [0 for i in range(len(loop2['op_arg_dat']))];
167   noOfSameDatAndArrayCases = 0;
168   for dataArg1 in loop1['op_arg_dat']:
169     for index in range(0,len(loop2['op_arg_dat'])):
170       if not matched[index] and not opArgDatMatch(dataArg1, dataArg2)
171          and not hasEqual[index]:
172         if opArgDatSameDataArray(dataArg1, loop2['op_arg_dat'][index]):
173           matched[index] += 1;
174           noOfSameDatAndArrayCases += 1;
175   propSimilarArgsLoop1 = noOfSameDatAndArrayCases/
176       len(loop1['op_arg_dat']);
177   propSimilarArgsLoop2 = noOfSameDatAndArrayCases/
178       len(loop2['op_arg_dat']);
179   propDiffArgsLoop1 = (len(loop1['op_arg_dat']) -
180       (noOfEqualArgs + noOfSameDatAndArrayCases))/
181       len(loop1['op_arg_dat']);
182   propDiffArgsLoop2 = (len(loop2['op_arg_dat']) -
183       (noOfEqualArgs + noOfSameDatAndArrayCases))/
184       len(loop2['op_arg_dat']);
185   if arch == Arch.CPU:
186     sameArgsWeighting = 10;
187     similarArgsWeighting = 5;
188     differentArgsWeighting = -0.2;
189     noOfArgsWeighting = -0.1;
190   else:
191     sameArgsWeighting = 10;
192     similarArgsWeighting = 5;
```

```
193       differentArgsWeighting = -0.2;
194       noOfArgsWeighting = -0.2;
195     complexity += (propEqualArgsLoop1 + propEqualArgsLoop2) *
196                    sameArgsWeighting * noOfEqualArgs + (
197                   (propSimilarArgsLoop1 + propSimilarArgsLoop2) *
198                   similarArgsWeighting * noOfSameDatAndArrayCases) + (
199                   (propDiffArgsLoop1 + propDiffArgsLoop2) *
200                   differentArgsWeighting *  (totalNoOfArgs -
201                   (noOfEqualArgs + noOfSameDatAndArrayCases))) + (
202                   totalNoOfArgs * noOfArgsWeighting);
203     return complexity;
204
205   def opDatArgsWeighting(CBRCase):
206     weighting = 0;
207     availableLoops = [];
208     for loop in CBRCase['op_par_loops']:
209       availableLoops.append(loop['loop_name']);
210     for pair in CBRCase['fusable_pairs']:
211       loop1 = getLoopInfo(pair['loop1'], CBRCase['op_par_loops']);
212       loop2 = getLoopInfo(pair['loop2'], CBRCase['op_par_loops']);
213       if opParLoopArrayMatch(loop1, loop2):
214         complexity =
215            calculateLoopFusionComplexity(loop1, loop2, CBRCase['arch']);
216         weighting += complexity;
217     return weighting;
218
219   def fusablePairsIntersection(fusablePairs1, fusablePairs2):
220     intersectingFusablePairs = [];
221     for pair1 in fusablePairs1:
222       for pair2 in fusablePairs2:
223         if pair1 == pair2:
224           intersectingFusablePairs.append(pair1);
225     return intersectingFusablePairs;
226
227   def opParLoopsIntersection(op_par_loops1, op_par_loops2):
228     intersectingOpParLoops = [];
229     for op_par_loop1 in op_par_loops1:
230       for op_par_loop2 in op_par_loops2:
231         if op_par_loop1 == op_par_loop2:
232           intersectingOpParLoops.append(op_par_loop1);
233     return intersectingOpParLoops;
234
235   def opArgDatComplexity(loop, arch):
236     complexity = 0;
237     dataSameArrayAccesses = [];
238     for dataArg in loop['op_arg_dat']:
239       found = False;
240       for index in range(0, len(dataSameArrayAccesses)):
241         if dataArg['name'] == dataSameArrayAccesses[index]['name'] and (
```

```
242           dataArg['indir_array'] ==
243           dataSameArrayAccesses[index]['indir_array']):
244         dataSameArrayAccesses[index]['occurrences'] += 1;
245         found = True;
246     if not found:
247       dataArrayAccess = {
248                         'name' : dataArg['name'],
249                         'indir_array' : dataArg['indir_array'],
250                         'occurrences' : 1
251                         }
252       dataSameArrayAccesses.append(dataArrayAccess);
253
254   noOfDifferentDataArrayArgsComplexity = 1;
255   if arch == Arch.GPU:
256     noOfDifferentDataArrayArgsComplexity = 1.1;
257   complexity +=  noOfDifferentDataArrayArgsComplexity
258                         * len(dataSameArrayAccesses);
259
260   totalNoOfArgsComplexity = 0.25;
261   if arch == Arch.GPU:
262     totalNoOfArgsComplexity = 0.5;
263   complexity += len(loop['op_arg_dat']) * totalNoOfArgsComplexity;
264   return complexity;
265
266 def similarityEstimation(CBRSystem, unmatchedCase):
267   noOfProperties = 6;
268   archWeight = 5;
269   weightedProperties = [ [ 0 for i in range(noOfProperties) ]
270                                     for j in range(len(CBRSystem)) ];
271   for index in range(len(CBRSystem)):
272     weightedProperties[index][CBRSystem[index]['case']['arch']] +=
273       archWeight * CBRSystem[index]['occurrences'];
274     weightedProperties[index][4] +=
275       fusableLoopsWeighting(CBRSystem[index]['case']) *
276       CBRSystem[index]['occurrences'];
277     weightedProperties[index][5] +=
278       opDatArgsWeighting(CBRSystem[index]['case']) *
279       CBRSystem[index]['occurrences'];
280
281   # normalize results
282   for index in range(len(CBRSystem)):
283     for prop in range(noOfProperties):
284       weightedProperties[index][prop] /= sum(weightedProperties[index]);
285
286   # here calculate the weight of the intersection
287
288   noOfIntersectingProperties = 3;
289   weightedIntersection = [[ 0 for i in range(noOfIntersectingProperties) ]
290                                       for j in range(len(CBRSystem)) ];
```

```
291   maxWeight = 0;
292   maxIndex = 0;
293   maxList = [];
294   for index in range(len(CBRSystem)):
295     if checkComparable(unmatchedCase['case'], CBRSystem[index]['case']):
296       intersectingArch = Arch.ANY;
297       if unmatchedCase['case']['arch'] == CBRSystem[index]['case']['arch']:
298         weightedIntersection[index][0] +=
299              weightedProperties[index][CBRSystem[index]['case']['arch']];
300         intersectingArch = unmatchedCase['case']['arch'];
301
302       intersectingCase =  {
303                           'arch' : intersectingArch,
304                           'fusable_pairs': fusablePairsIntersection(
305                               CBRSystem[index]['case']['fusable_pairs'],
306                               unmatchedCase['case']['fusable_pairs']),
307                           'op_par_loops': opParLoopsIntersection(
308                               CBRSystem[index]['case']['op_par_loops'],
309                               unmatchedCase['case']['op_par_loops'])
310                           }
311
312       weightedIntersection[index][1] +=
313           fusableLoopsWeighting(intersectingCase) *
314           weightedProperties[index][4];
315       weightedIntersection[index][2] +=
316           opDatArgsWeighting(intersectingCase) *
317           weightedProperties[index][5];
318       if sum(weightedIntersection[index]) > maxWeight:
319         maxWeight = sum(weightedIntersection[index]);
320         maxIndex = 0;
321         maxList = [];
322       if sum(weightedIntersection[index]) == maxWeight:
323         maxIndex += 1;
324         maxList.append(CBRSystem[index]);
325
326     if maxIndex > 0:
327       for index1 in range(maxIndex-1):
328         for index2 in range(index1+1,maxIndex):
329           if maxList[index1]['occurrences'] <
330              maxList[index2]['occurrences']:
331             aux = maxList[index1];
332             maxList[index1] = maxList[index2];
333             maxList[index2] = aux;
334
335   if len(maxList) > 0:
336     return maxList[0];
337
338   return None;
339
```

```
340 def bestCaseMatch(CBRSystem, unmatchedCase):
341   tempCase = caseLookup(CBRSystem, unmatchedCase);
342 
343   if tempCase == None or not equals(tempCase, unmatchedCase):
344     tempCase = similarityEstimation(CBRSystem, unmatchedCase);
345 
346   return tempCase;
347 
348 def checkBestMatch(CBRSystem, newCase, bestMatch):
349   maxComplexity = 0;
350   loopFusionComplexity = [];
351   wantedFusion = None;
352   for pair in newCase['case']['fusable_pairs']:
353     loop1 =
354        getLoopInfo(pair['loop1'], newCase['case']['op_par_loops']);
355     loop2 =
356        getLoopInfo(pair['loop2'], newCase['case']['op_par_loops']);
357     if opParLoopArrayMatch(loop1, loop2):
358       complexity =
359            calculateLoopFusionComplexity(loop1, loop2,
360                newCase['case']['arch']);
361       fusionComplexity =  {
362                           'fusion' : pair,
363                           'complexity' : complexity
364                           }
365       loopFusionComplexity.append(fusionComplexity);
366       if complexity > maxComplexity:
367         maxComplexity = complexity;
368 
369   threshold = 0;
370   if maxComplexity > 0:
371     # we have a good loop fusion
372     # now we check if the bestMatch does it;
373     wantedFusion = None;
374     for fusion in loopFusionComplexity:
375       if fusion['complexity'] == maxComplexity:
376         wantedFusion = fusion['fusion'];
377     print wantedFusion;
378     if bestMatch != None and
379        checkComparable(newCase['case'], bestMatch['case']):
380       loopsToFuse = bestMatch['solution']['loops_to_fuse'];
381       if loopsToFuse.index(wantedFusion) != -1:
382         return [True, bestMatch];
383       else:
384         if newCase['case']['arch'] == Arch.CPU and
385            bestMatch['solution']['op_warpsize'] == 1:
386           return [True, bestMatch];
387   else:
388     # we have no loop fusions
```

```
    if bestMatch != None and
       checkComparable(newCase['case'], bestMatch['case']):
      if len(bestMatch['solution']['loops_to_fuse']) == 0:
        return [True, bestMatch];
      else:
        if newCase['case']['arch'] == Arch.CPU and
           bestMatch['solution']['op_warpsize'] == 1:
          return [True, bestMatch];

  # else: we have a better case and we create it
  op_warpsize = 1;
  if newCase['case']['arch'] == Arch.GPU:
    op_warpsize = 32;

  overallMaxComplexity = 0;

  for loop in newCase['case']['op_par_loops']:
    complexity = opArgDatComplexity(loop, newCase['case']['arch']);
    if complexity > overallMaxComplexity:
      overallMaxComplexity = complexity;

  if maxComplexity > overallMaxComplexity:
    overallMaxComplexity = maxComplexity;

  complexityThresholdForDiffValuePartBlkSize = 4;

  temp_block_size = 128;
  temp_part_size = 128;
  referenceValue = 4;
  adjustmentFactorCPU = 8;
  adjustmentFactorOther = 32;
  print overallMaxComplexity;
  if newCase['case']['arch'] == Arch.CPU:
    temp_block_size = math.log(overallMaxComplexity * referenceValue)
      * adjustmentFactorCPU;
  else:
    temp_block_size = math.log(overallMaxComplexity * referenceValue)
       * adjustmentFactorOther;

  diffArray = [];
  for val in possible_values_blk_part_size:
    diffArray.append(math.fabs(temp_block_size - val));
  minDiff = min(diffArray);
  for index in range(len(possible_values_blk_part_size)):
    if (diffArray[index] == minDiff):
      temp_block_size = possible_values_blk_part_size[index];
  if overallMaxComplexity <
     complexityThresholdForDiffValuePartBlkSize
    and temp_block_size *4 <= 512:
```

```
438       temp_part_size = 4 * temp_block_size;
439     elif temp_block_size * 8 <= 512:
440       temp_part_size = 8 * temp_block_size;
441     elif temp_block_size < 512:
442       temp_part_size = 2 * temp_block_size;
443     else:
444       temp_part_size = temp_block_size;
445 
446     newSolution = {
447                   'loops_to_fuse' : wantedFusion,
448                   'op_warpsize' : op_warpsize,
449                   'block_size' : temp_block_size,
450                   'part_size' : temp_part_size
451                   }
452 
453     newBestMatch =  {
454                     'case' : newCase['case'],
455                     'solution' : newSolution,
456                     'occurrences' : 1
457                     }
458     return [False, newBestMatch];
459 
460 def retrieve(CBRSystem, newCase):
461     bestMatch =  bestCaseMatch(CBRSystem, newCase);
462 
463     # failsafe no 1 - we need to see if this is really a good option.
464     # this situation might have not been encountered
465     [betterMatchFound, betterMatch] =
466       checkBestMatch(CBRSystem, newCase, bestMatch);
467     return betterMatch;
468 
469 def reuse(bestCase, newCase):
470     newCase['solution'] = bestCase['solution'];
471     return newCase;
472 
473 def retain(CBRSystem, solvedCase):
474     return checkExistsAndIncr(solvedCase, CBRSystem);
475 
476 def CBRInit(CBRSystem, trainingCases, results):
477     for index in range(0, len(trainingCases)):
478       CBRSystemCase = {
479                       'case' : trainingCases[index],
480                       'solution' : results[index],
481                       'occurrences' : 1
482                       };
483       CBRSystem = checkExistsAndIncr(CBRSystemCase, CBRSystem);
484     return CBRSystem;
485 
486 ...
```

```
487
488 [trainingCases, results] = retrieveTrainingData();
489 CBRSystem = CBRInit(CBRSystem, trainingCases, results);
490
491 # create vector of properties - the case
492
493 newCase = {
494           'case' :  {
495                     'arch' : default_arch,
496                     'fusable_pairs' : fusable_pairs,
497                     'op_par_loops' : op_par_loops
498                     },
499           'solution' : None,
500           'occurrences' : 1
501           }
502
503 # call machine learning to retrieve best result for the current case
504
505 bestCase = retrieve(CBRSystem, newCase);
506
507 solvedCase = reuse(bestCase, newCase);
508
509 print solvedCase['solution'];
510
511 CBRSystem = retain(CBRSystem, solvedCase);
```