University of Bath

PHD

Auto-tuning compiler options for HPC

Jones, Jessica

Award date:
2019

Awarding institution:
University of Bath

Link to publication

Alternative formats
If you require this document in an alternative format, please contact:
openaccess@bath.ac.uk

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 11. May. 2021

Citation for published version:
Jones, J 2019, 'Auto-tuning compiler options for HPC', Ph.D., University of Bath.

Publication date:
2019

Document Version
Publisher's PDF, also known as Version of record

Link to publication

Publisher Rights
GNU GPL

University of Bath

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 16. Jul. 2019

Auto-tuning compiler options for HPC

submitted by

Jessica R. Jones

for the degree of Doctor of Philosophy

of the

University of Bath

Department of Computer Science

August 2018

COPYRIGHT

Attention is drawn to the fact that copyright of this thesis rests with the author. A copy of this thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with the author and that they must not copy it or use material from it except as permitted by law or with the consent of the author.

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signature of Author . . . . . .

Jessica R. Jones

ACKNOWLEDGEMENTS

> I must frankly own, that if I had known, beforehand, that this book would have cost me the labour which it has, I should never have been courageous enough to commence it.
>
> Isabella Beeton

There are many people who should be thanked for their patience, understanding and time. These include, but are not limited to, my supervisors, James Davenport, John ffitch and Guy McCusker, my partner, Ed White, my sisters, Eve Jones, Zoë Jones and Rebecca Bonome, and my friends (in no particular order), Martin Brain, Florian Schanda, David Stewart, Mark Cahill, Alaric Snell-Pym, Anita Faul, Mesar Hameed and Ben Pring. All have helped in various ways, from proof-reading to giving advice at odd hours on Sundays, to bringing me cups of tea and making sure some shred of sanity still remains at the end of this. I could not have done it without your support.

The HPC support staff at the University of Bath and at the University of Southampton are also due thanks; without the systems they are responsible for, this work would not have been possible.

I would also like to thank Jason Ansel, author of OpenTuner, and Clint Whaley, author of ATLAS, for the advice they gave both in person and by email. They both helped me to avoid wasting precious time, and made my life a little bit easier.

Considerable thanks are due also to my examiners, who gave up their time to read my work and make suggestions for its improvement. Having acknowledged that I am probably one of the many who "do not know how to write" [121], I know that their insightful comments and thought-provoking questions have greatly aided in achieving the clarity this document requires. I hope I have done justice to their hard work.

Last, but not least, I must thank my colleagues at Cray UK Ltd, who put up with my ramblings and kindly agreed to cope without me for weeks while I battled to finish this. The contribution of time is more valuable than you know.

Contents

List of Figures

List of Tables

LIST OF ALGORITHMS

List of Prototypes

Abstract

Many of the principal high-performance libraries are written in assembly language. When a new CPU architecture appears on the market, it takes time for the developers of high performance libraries to gain access to it and to port their kernels to that system. During that period performance for a wide range of applications suffers.

To provide a suitable middle ground, the author proposes using a tailored subset of the optimization options built into the available compilers. The available options are not ideal for all software programs on all architectures, and in the general case compilation times need to be kept short. However, scientific computing kernels are so heavily used that an expensive one-off compilation can be afforded.

Searching the large number of available compiler flags by hand would be extremely time consuming, and difficult to do without an intimate understanding of the hardware, the compiler of choice and the software being compiled. Instead, this search is best automated, and that is what the author has attempted to do in the form of a tool referred to as `OptSearch`.

This tool reads in a number of options from a configuration file: These include the list of compiler flags to search through, the application to be compiled, a clean-up routine, and two sets of tests to run; one for accuracy, one for performance.

After investigation of a number of optimisation algorithms, a robust search method (PSO [49]) is selected and used as a number of the compilations are liable to end in an error from the compiler. The end result is a set of compiler flags that deliver better performance than the defaults for that particular piece of code on that hardware. While this will not rival expert hand-coding, it is observed to deliver significant performance improvements over using `-O3`.

To date, `OptSearch` has been used with the reference BLAS [117] and the HPL benchmark [136]. In the future the author intends to target other libraries fundamental to scientific computing.

CHAPTER 1

INTRODUCTION

> It's not overstating the case to say that software is vital to research. If we were to magically remove software from research, 7 out of 10 researchers would be out of a job.
>
> Simon Hettrick [80]

Most modern scientists make use of high performance computers (HPCs) for the running of computational models or data mining and statistical analysis. The greater the speed or throughput of the machine, the more detail the model can contain and the larger the dataset being analysed can be. HPC is a vital part of modern science, and, with over 90% of UK researchers dependent on software to facilitate their work [80] [104], the importance of good, maintainable software to research cannot be over stated.

Users of HPC depend heavily on advances both in hardware and software to achieve the staggering speeds of computation announced in press releases and in the TOP500 [122] list. Unfortunately, these advances rarely happen at the same rate. In practice, it is common for high performance libraries, required by the many applications in use on HPC systems, to take several months to deliver the same level of performance on a newly introduced system that were seen on the previous generation.

These libraries rely on hand-coded routines, generally written in assembler, and require a great deal of skill and tuning to deliver such high efficiencies. There can also be difficulties in library developers gaining access to a new microarchitecture, leading to further delays before the high efficiencies seen in the TOP500 list (often with a R_{max} or achieved floating point throughput of 90% or more of the theoretical peak, R_{peak}) can be realised. Debugging is difficult and portability depends on having a pre-written kernel for the architecture being targeted, or requesting that the developers support it (requiring that they have access to a suitable machine on which to work).

Often, scientific software developers will make adjustments to their software to increase performance. Many of these adjustments reduce the portability of the application in which they are made, since they frequently depend on particular compilers, operating systems and hardware to work as intended. Moving to another machine may result in software that does not compile, or behaves unpredictably, depending on the changes that were made.[1]

[1] In introductory courses on using HPC systems, the author frequently welcomes users to the 1960's, although in truth things are better now as standards have meant far less variation between machines.

Unfortunately, many developers of scientific software work on a single project for only a few years before moving on. It is left to the next person, usually a PhD student like their predecessor, to pick up where they left off and advance the work forward, often moving to a newer machine in the process. Maintainability and portability of the software could be argued to be of high importance in such situations, but these things are rarely priorities in practice, compared with a paper or thesis deadline.

This problem has been recognised by others, and in recent years efforts have focussed on addressing it through various means, including community codes (some with well defined coding standards), the Research Software Engineer (RSE) movement and efforts of the Software Sustainability Institute (SSI) [2], among others. These efforts have helped to alleviate the problem in some of the more established disciplines in HPC, but it is an ongoing battle with a long way to go before the problem can be considered solved.

It remains the case for many people both in and outside of scientific computing that if a given program compiles and runs, this is good enough. Little thought is given to the flags passed to the compiler at compile time, whether these are provided by the user, or more commonly, by the Makefile supplied with (or generated by the build process of) the software. However, where it is important to get as much efficiency (however that is defined) out of a program as possible, it may pay dividends to study the compiler manual and customise the flags accordingly. To realise this performance benefit, the individual responsible for choosing and customising the chosen flag/parameter set for an application must have an intimate knowledge not only of the effects of the chosen flags and their values, but also of the target hardware and the software being compiled. A sufficiently high level of familiarity with one is uncommon; with more than one of these it is rare.

Administrators of supercomputers are probably more interested than most in having applications on the machines they manage run as fast as possible[2], yet without having the time and the necessary depth of knowledge, (even) they cannot choose the most effective compiler flags for each program, nor for each compiler. The author, an experienced supercomputer administrator of many years, is aware that it is common for the administrators of such systems to spend as much as 6 months to a year rebuilding the set of software required by the users whenever a new machine comes into production. With the broad range of software packages and libraries typically required by the research community at a single institution (a subset of which can be seen in the ARCHER documentation [51]), it is unlikely that any one individual could be familiar with them all to the extent necessary to choose the best compilation options in every case.

Students of scientific computing courses are often taught arcane ways to optimize their code. Some of these optimizations are things that we might reasonably expect a compiler to be able to do, such as loop unrolling, tiling and jamming, and code in-lining. Compilers, they are told[3], are not good enough to do this for you, at least not beyond a very basic level. These alterations to what was a relatively simple piece of code are necessary to achieve performance [167], and have been for some time[4], even though they may make it unreadable and difficult to debug and maintain. The abundance of such difficult to comprehend software is a growing problem in these times of heightened awareness of the insecurities of both hardware and software, when peer review could be considered critical. They also affect the ease of porting the

[2] or, increasingly in these times, as energy efficiently as possible

[3] On one course, taught by an employee of a well known chip manufacturer, the author was told this despite the chip manufacturer investing heavily in their own compilers and high performance libraries

[4] The author's supervisor wrote his first piece of self-inlining machine code to improve FORTRAN performance in 1971

code from one architecture to another, as further code changes are necessary to maintain performance, and a change that is beneficial with one combination of compiler and target architecture may be detrimental to, or even entirely incompatible with, another.

That code must be altered in this way is something that has been taught to more than one generation. The author has heard or read it stated (and re-stated) for more than a decade. The life time of the average scientific code is generally greater than a single CPU architecture[5], making a more portable method highly desirable.

1.1 Motivation

> Portability might matter more than mere performance. That is, you can sacrifice probably a few percent of extra performance if you get on to better portability and sustainability of your developments.
>
> Jean-Philippe Nominé, CEA [97]

Many scientists find their work is limited in how much data can be processed, how large a simulation can be run or how finely detailed that simulation. These limitations are often described as being due to the availability of the hardware, but this is not necessarily the only limitation. Supercomputing centres are usually quick to invest in hardware, but few show a willingness to invest in the software their facilities are created to support.

Performance portability is a significant challenge for the average user of HPC. Anything that allows them to move their work to the next machine without requiring them to acquire new skills or expend effort that might otherwise be used for research is likely to be well received. As explained in [7, page 31]:

> "Users are overwhelmingly concerned about the challenge of performance portability. The high development and maintenance effort required to tune to multiple platforms is considered a large burden, taking time and resources that might otherwise be spent on other aspects of the projects. As a result, developers may either limit the number of platforms their codes are ported to or limit how well their codes are optimized for specific platforms of interest."

Even on established hardware, many of the most popular applications fail to deliver even 10% of theoretical peak floating point performance (R_{peak}) on the machines they are running on [20] [24][6] [47, slide 22].If efficiency could be increased, throughput or simulation detail could be increased without needing to invest in more and better hardware.

Efficiency does not need to be increased greatly to have a significant impact on machine throughput. In [17] the case is made for investing in 2 months of an expert's time in return for a 5% improvement in performance. Despite the extremely conservative estimate of 5%, and the demonstration of its worth even with this relatively low return, in practice it is more common for HPC sites to invest in hardware. This is perhaps better attributed to financial constraints placed on such sites, rather than simply an indication of how software investment is viewed. If not designed with portability in mind, software improvements are unlikely to be portable. Thus the investment cannot continue to offer returns beyond the lifetime of the current production machines.

[5] The UK Met Office typically buys a new computer for production purposes every three years, but embarks on a major software re-write every 20 years [23].

[6] In point of fact, [24] is now more than 10 years old, and the problem appears to be getting worse.

While every scientific software developer (and user) is likely to know about the pre-defined optimization levels of compilers (`-O2` and `-O3` being the most commonly used), there is less understanding of what these do in practice. Compilers generally have a number of flags for tuning optimization. GCC [154] version 7.3.0 has over 400 flags for this purpose (the exact number varies depending on the target architecture and how the compiler itself was built), yet the pre-defined optimization levels will only equate to a subset of these.

There are also many who do not realise that the pre-defined optimization levels, such as `-O2`, are not equivalent on all compilers. Rather, they are the result of choices made by the compiler developers, who must make many assumptions about the typical usage of their software and the environment in which it will run. GNU, whose compilers are widely used for open source software, including Linux distributions, must worry about portability far more than Intel, who need only support one architecture, x86, and are free to prioritise performance on a smaller list of hardware – their own.

Unfortunately, very few people have the time or the knowledge to go looking through the compiler manuals, and sometimes the source, for the compiler options available to them. They have even less time available to spend searching for the best combination, something that can take a considerable amount of time, even for simple programs. Yet those who have taken the time to do the search by hand have already observed significant gains in performance, and these are often appreciated by HPC users (see for example the email in appendix D). This demonstrates that compilers, used correctly, can deliver the needed improvements in at least a handful of cases. The difficulty is in knowing how best to instruct them. The search space is so vast that a naive search is simply not practical.

Fortunately, it has been shown that it is possible to find "good" combinations of compiler optimizations programmatically, for example in [131], though [6] highlights the difficulty of picking an appropriate search algorithm.

Auto-tuning has been turned to in various guises in an attempt to solve the problem of performance portability, giving rise to a number of possible auto-tuning approaches. An overview of these is suggested in [13, table 1, page 3]. Surprisingly, the approach suggested by this project, of auto-tuning (or automatically assisted tuning of) the selection of compiler flags, does not appear in this table. This type of approach would be best applied during porting: When a new machine has been commissioned and delivered but has not yet entered full production (that is, it is not yet being heavily used).

1.2 Aims of this thesis

At present, tuning the performance of programs is something done only by those few who have the time, aptitude and the knowledge to do it. Much of it involves taking those sections of the code identified to be most critical to performance and re-writing them in assembly code. To date, it has not been possible to replace such work with an automated process, and it requires both time and extensive familiarity with both the algorithms and the target hardware. As a result, there is a noticeable delay, usually in the order of 12-24 months, before a high performance library for a new CPU architecture (or microarchitecture) achieves the performance seen on existing, established hardware. Prior to that, a compatible library (other than a reference implementation) may not even be available.

[34] classified scientific applications by their broad type and application. From this list of 7 (since expanded to 13 [133]), dense linear algebra has been chosen as the first topic for thorough investigation. The author considers that this is the most well understood, with work on

optimizing the Basic Linear Algebra Subprograms (BLAS) [117], in particular double precision (dense) general matrix multiplication (`DGEMM`), having been driven largely by the TOP500 list [122], and the commercial implications thereof, for slightly more than two decades.

The BLAS libraries delivering the highest performance are well known, and depend heavily on kernels (sometimes in a parameterised form) written by hand in assembler or machine code for particular processors. One side-effect of this is the delay in support for new hardware. Hardware that is not available, or not of interest, to the authors of the libraries is rarely supported by them. For example, [184], [172] and [175], give a few occasions where difficulties in obtaining hardware has delayed support.

In this thesis, an alternative approach is proposed, inspired by the author's success in hand-tuning the selection of compiler flags of commonly used libraries (see appendix D for example). Bischof et al justified a high investment in hand-optimization to gain a 5% improvement in performance [17], suggesting a target for success for an automated approach, something which will surely require a lower future investment.

The questions to be answered here are:

RQ1 Is machine code necessary for performance?

RQ2 Is performance-portability achievable?

RQ3 What is the economic value?

These questions are equally valid for libraries and applications. In this thesis, these questions will be answered in the context of libraries, rather than applications. This is to keep the approach both simple and to affect as great a number of scientific applications as possible. Well established libraries frequently have thorough test suites, which allow the builder of the library to check that numerical integrity has not been compromised. This is difficult to do for applications, which typically perform a number of functions depending on the input data and usage. It is not clear that these research questions can be answered for an application such as DL_POLY [165] outside of the small test cases. In addition, by targeting libraries it is possible to avoid the problem of how to deal with areas of the code that have been written in assembly language or using architecture-specific compiler intrinsics, as is the case in applications such as GROMACS [15] (as described in [144]).

1.3 Document overview

The reader is first presented in chapter 2 with an overview of how the supercomputing landscape has changed since the 1960s, increasing in complexity through to the present day in order to increase processing speed and throughput. Many of the challenges faced in attempting to make use of these advances today are constrained by historical decisions.

Discussion moves on with chapter 3 and an overview of similar work in this field, with a note about the difficulties of understanding default compiler optimization levels.

In chapter 4, attention is drawn to the author's early work investigating the workings of two high performance implementations of the BLAS, and of difficulties arising from the purchase of new and emerging hardware.

This is followed in chapter 5 by an analysis of the requirements of a software auto-tuning tool through iterative prototyping and discussion of this project's high-level use case.

In chapter 6, a study on the convergence of a common benchmark is presented; a necessary investigation for automated judgement of such to be successful. This study was carried out in

parallel with the creation of early prototypes (discussed in section 5.2.2), and used to inform later design decisions.

Chapter 7 contains a discussion of the design specification and lists the decisions made as a result of the prototyping work, while chapter 8 attempts to explain some of the difficulties that arose during implementation. Results are given in chapter 9 and a critical assessment of these can be found in chapter 10.

Several opportunities for further work are discussed in chapter 11. The document concludes in chapter 12 with answers to the research questions listed above and their implications for the wider HPC community.

CHAPTER 2

THE LANDSCAPE

> Some changes come from unexpected directions. Take Moore's Law, for example. Moore's Law had a far greater impact on the latter third of the 20th century than the lunar landings of the Apollo program (and it was formulated around the same time), but relatively few people know of it. Certainly, back in 1968 nobody (except perhaps Gordon Moore) might have expected it to result in the world we see today ...
>
> Charles Stross [156]

The landscape of scientific computing has changed greatly over the past decade, and its use and popularity has gained ground across all scientific disciplines, especially in recent years. A proper treatment of the history of computing, and especially as it relates to modern supercomputers, would require its own book (or series of them). This chapter attempts to give a stratospheric overview of the key innovations that have brought computing, and particularly supercomputing, to where it is today.

Standards in hardware (e.g. IEEE 754 [72]) and software have helped to make moving software from one machine to another less painful than it once was, both in terms of it running at all and in terms of its running "correctly". Language [25] [59] and library [60] standards have also served to make life considerably easier for those developing applications.

Despite the success of standardization, achieving performance portability is still beyond the reach of most, and platform-specific optimizations tend to result in unmaintainable (or very difficult to maintain) code. Some, such as the authors of Intel's Math Kernel Library (MKL) library, have gone the route of "fat binaries" and use CPU dispatchers to achieve the appearance of performance portability between their own chips.

2.1 Computational Hardware

The first commercially successful computer was the UNIVAC 1, which began shipping in June 1951 with a list price of $250,000 [78], a significant amount of money at the time.[1]

[1] In the December 1957 issue of the New Scientist, the University of Southampton offered the princely sum of £250 a year for a "computer operator" for the Ferranti Pegasus they were in the process of acquiring.

IBM soon followed suit, producing several computers through the 1950s, 1960s and 1970s (and beyond). Until the mid-1980s it was not yet normal for computers even from the same manufacturer to share a common architecture. It was not until 1964, when the IBM 360 debuted, that designers began to realise that machines that shared a common machine language could be cheaper to produce. This was found to benefit not only their own programmers, but be popular with customers, who had previously had to invest considerable effort to port their programs from one machine to another.

This realisation of the importance of portability is probably responsible for a large part of the success of these early IBM mainframes. As a consequence of this wide adoption, many of the IBM engineers' decisions affect us almost half a century later[2].

2.1.1 Purpose-built supercomputers

The most well-known supercomputer is the Cray-1 (1976), but it was not the first, as that honour is usually given to Control Data Corporation's (CDC) 6600 (1964). However, several machines preceding both of these were used for scientific computing. One such example was the Ferranti Pegasus (1956) installed at the University of Southampton in 1957. This machine was used in the (structural) design of the Sydney Opera house[3]

The 1960s brought more "giant machines", ICT boasting in Autumn 1967[4] of a 1906A capable of one instruction per microsecond and a whopping 64k words of main memory. Computer mainframes of this type, while impressive, could not match the performance of vector supercomputers, which quickly became the computers of choice for those working in scientific computing.

The first generation of vector supercomputers included the Illiac-IV, Texas Instruments' (TI) ASC and Control Data Corporation's (CDC) Star-100, all produced during the 1960s. The Illiac-IV did not prove to be very popular, with only a few being installed, while the Star-100 and ASC were more successful. Both of these systems could achieve a remarkable 40 megaFLOPS in the hands of a skilled operator.

The second generation of vector supercomputers brought with it the Cray-1. With Seymour Cray having learned much from his days at CDC, it shared some of the features of the CDC-6600/7600 systems, though with many improvements. Until the early-1980s, it was considered to be the fastest supercomputer ever built, capable of achieving 160 megaFLOPS if programmed efficiently. Quite an achievement for the time.

The Cray-1 benefited not only from its multiple pipelines and vector processor design, but also from its accompanying vectorising compiler, which supported an extended FORTRAN. Soon after its release, CDC and Fujitsu continued the trend with the CDC Cyber-205 (descended from the Star-100) (1982) and Fujitsu F230-75 APU (1977), VP-200 (1983) and later machines. These three were joined by NEC and Hitachi in Japan, and by Convex, Parsytec, Silicon Graphics and MasPar in the US, and for almost two decades vector supercomputers dominated the TOP500 [122] list.

All of these systems utilised multiple pipelines for concurrent vector processing, something that is still relied on by modern CPUs to increase floating point throughput and thus

[2]When they chose to move from 24-bit to 32-bit addressing, but did not immediately make use of it for performance reasons, programmers quickly began to make use of those unused extra bits. Modern compilers still check the values of those bits, to ensure that this "trick" is not being employed on modern machines.

[3]Sadly acoustic modelling did not make an appearance until some years later.

[4]A former employee, writing in the Winter 2017 issue of the journal of the Computer Conservation Society, notes that there was not one in operation until a year later, so ICT were exaggerating in their press release

performance. To facilitate access to larger pools of memory, computer designs moved from predominantly using 24-bit addressing to 32-bit addressing, while caches allowed those accesses to happen with relatively low latency. Good compilers were as essential then as they are now, and most programmers adopted one of two high-level languages that still dominate HPC software now: Fortran, or the (at the time) trendier C.

However, despite this movement towards uniformity, CPU architectures have diverged in other ways, and recent new-comers to the scientific computing world have called for the adoption of new approaches and techniques. Those working on software for these systems now have to consider variation in clock speed between individual cores within a chip [91], as well as differing clock speeds between "identical" chips that are still deemed to be within specification [113].

More recently, it has become necessary for both programmers and (especially) compiler writers to make adjustments for differences in floating point arithmetic between chips from the same manufacturer [95]. This is something that has not been a concern for around 2 decades, and is requiring the latest generation of programmers to re-discover or reinvent many of the tricks of their predecessors.

2.1.2 Floating point hardware

Until 1985[5], moving from one machine to another entailed a great deal of work, especially in checking that the outputs were as expected. This was not only due to the differences in operating systems, but also the large variation in architecture. In fact, it was not until some years after 1985 that manufacturers settled, largely, on machines with the same byte size. Now only a few chips exist that do not use 8-bit bytes.

In the 1960s, bytes of 6-bits were more common, and there was a great deal of variety throughout the 1960s into the 1990s. Some early machines, such as the UNIVAC 1050, had variable length words, and not all machines used binary. Those that did were not all using the same binary representation. It certainly made life more challenging for the programmer. Now, it seems that only specialist architectures such as XAP, used for ASICs, use a different size fundamental unit (in this case, 16-bit, 24-bit or 32-bit).

2.1.3 Multiply-Accumulate

Multiply-Accumulate (MAC) operations are variously known as multiply-add fused (thanks to IBM), Fused Multiply-Add (FMA) or fused multiply-accumulate (FMAC) when performed with a single rounding (as opposed to 2). They are described in the 2008 edition of IEEE754 [72] and have been supported in the C standard since C99 [26]. They have been implemented in hardware for much longer, at least since VAX implemented their `POLY` instruction in 1977, though before its inclusion in IEEE 754 the exact implementation varied by machine.

These instructions allow the computation of the product of two numbers and its addition to an accumulator to be computed in one instruction. In a modern CPU with a dedicated MAC unit, this instruction is performed in a single cycle and may be combined with one or more Single Instruction, Multiple Data (SIMD) instructions. MAC operations are also used to increase floating point throughput in modern Graphics Processing Unit (GPU)s, which have much in common with early vector processors.

[5] In reality, it was a little while before the IEEE 754 standard was adopted widely by hardware manufacturers

2.1.4 Vectorisation

Vector or SIMD instructions involve the application of one operation to vectors of values. In most current implementations, the hardware design limits the length of these vectors to one of a set of fixed sizes. This type of instruction was utilised by many of the early supercomputers and their predecessors throughout the 1970s. However, most of those working on software today are likely to have first encountered SIMD instructions in the form of x86 MMX, which arrived on the scene in the late 1990s. Coincidentally, this was about the same time that dedicated vector processors, such as those designed by Cray Research, were on the way out.

Unlike the processors of vector supercomputers, the MMX instructions are a fixed size, using 8 64-bit wide registers to apply a single operation to 2 64-bit integers, 4 16-bit integers or 8 8-bit integers in one instruction. The registers are named `MMX0` through `MMX7`, making them easy to spot in generated assembler.

Since MMX instructions were introduced, many others (SSE, SSSE, 3dnow, etc) using wider and wider vector registers have appeared over the years. The latest available version in the x86 instruction set is AVX-512. As the name suggests, the vector registers used are 512-bits wide. Wider SIMD instructions are expected to follow soon.

More recently, Arm Ltd have published processor designs using variable length vector instructions[6] (referred to as scalable vector extensions, or SVE), which includes a form for complex numbers [9]. The latter is likely to be of interest to both engineers and chemists, since prior to that, operations on complex numbers could not be (efficiently) vectorised and anyone reliant on the Fourier transform for their work would see little benefit from the increased throughput of SIMD instructions. While the x86 instruction set does include some support for complex numbers, it is limited to a few SSE4 instructions at present.

2.1.5 Accelerators

While use of GPUs has garnered a lot of attention during the last two decades, the use of additional hardware to provide higher computational throughput is not unprecedented. "Attached array processors", such as the IBM 3838 were available during the 1960s and 1970s, allowing more general purpose machines, both mainframes such as the IBM 308X series, and minicomputers such as the DEC PDP-11 and VAX 11-series, to be used for scientific work. These attached processors, as the name implies, could not be used alone, just as a modern GPU cannot be used in isolation. They also relied upon multiple pipelines and vectorisation.

The two most popular attached processors in the early 1980s were Floating Point Systems's (FPS) AP-120B and FPS-164. These worked very similarly to today's GPUs, providing a way to process large vectors or matrices at a faster rate than a standard minicomputer or mainframe, yet at a substantially lower cost than that of a purpose-built supercomputer. They could not deliver the same performance: FPS claimed a mere 12 megaFLOPS for the AP-120B.

Modern General Purpose Graphics Processing Unit (GPGPU) hardware has until recently tended to remain in card form, limited in bandwidth and subject to the latencies of the PCI-e BUS over which it must communicate with the rest of the machine. Some of the more recent designs, such as NVidia's P100, have moved to a CPU-like architecture, allowing them to benefit from increased bandwidth and reduced latencies for data movement. Other innovations have focussed on improving network bandwidth and latencies, allowing GPU cards to pass data between one another more directly.

[6] This produced much nostalgia among the older Cray engineers.

Users of GPUs often find it challenging to write software for them, particularly with the difficulty of debugging applications on something that uses direct memory access (DMA) and so accesses main memory independently of the CPU. Data movement is even more costly, and there is arguably greater variation in architecture even between cards made by the same manufacturer. However, performance gains are often seen if an appropriate algorithm exists, and several have been suggested, for example in [3]. Even so, performance portability is rarely seen in practice, as GPU hardware designs differ far more even between generations from the same manufacturer than do modern CPU designs. This has made them the target of many high performance library authors (for example [74] and [53]), as well as attempts to autotune or automatically generate kernels, particularly for stencil computations (as in [101]).

The continued popularity of these accelerators, which could be considered a type of vector processor, says something about the nature of the algorithms they are used for. It is not surprising that modern CPU manufacturers have begun to include support for ever-larger vector operations in their chip designs.

Specialised hardware

Although the area has seen interest for more than 40 years, Artificial Intelligence (AI), and Machine Learning (ML) in particular, interest in this area exploded in the early 21st century. This has given rise to investment in specialised hardware for the purpose. Some, such as Google [147] and Tensilica (since bought by Cadence) [86], have even gone so far as to focus their efforts on particular applications (in this case, TensorFlow [4]). Others, such as [87], are targeting broad classes of ML algorithms. Even the venerable IBM is investing in research in this area [123], propelled in large part by increased interest and funding from organisations such as DARPA.

These chips are not destined to take over from "traditional" CPU designs (at least, not yet), but are expected to be used in a similar way to the old attached-array processors and current GPUs. Due to the greater variation between designs and rapid speed of development, it is likely that many applications will continue to rely upon skilled programmers for performance benefits for some time to come.

2.1.6 Computational Speed and FLOPs

In the HPC world, performance is currently measured in floating-point operations per second (FLOPS), or rather gigaFLOPS[7] (thousands of millions of (IEEE `binary64`) floating-point operations per second), where an operation is any one of addition, subtraction, multiplication or division.

What effect do all these features have on the performance of a computer? Suppose we have a (somewhat simplistic[8]) model machine with FMA and AVX-512, executing one instruction per cycle. Then one cycle can operate on vectors of eight `binary64` numbers at a time, doing one FMA operation on each, hence sixteen floating-point operations in terms of raw GFLOPs. Thus a 2.5 GHz machine could be (and therefore will be) quoted as a 40 GFLOP machine, even though it might only be capable of doing 2.5 GFLOP when doing individual divisions.

Figures of this type could be considered misleading, and it is just one of many criticisms of the TOP500 [122] and the HPL [136] benchmark in particular ([44, slide 9]).

[7] Today, even desktop computers can be measured in GFLOPS; supercomputers today are typically rated in petaFLOPS, but benchmarks still output their figures in GFLOPS.

[8] Ignoring pipelines, warm-up, narrow vectors etc.

2.2 Interconnect

As those working in scientific and high performance computing know, computational speed is not the only, or even necessarily the best, indicator of good performance. Parallel applications, by their nature, depend on communication between processes. Some applications may spend more than half of their execution time in calls to communication libraries, with Message Passing Interface (MPI) being the most commonly used currently.

Good communication performance depends on several factors, particularly network bandwidth and latency. A machine with high computational performance but high network latencies is liable to result in application processes stalling as they wait for the next message to arrive or be received. Not every algorithm can be expressed in an asynchronous form, and those that can often still rely on some parts happening in a particular order. Interconnect has thus been fundamental to supercomputer performance for some time, especially for real applications (as opposed to artificial benchmarks).

Early supercomputers had very simple internal interconnect. The Cray 1, which was composed of a series of boards or "modules", depended on a great many pairs of wires carefully measured to precise lengths to guarantee timings. A single arithmetic unit was effectively built from several modules, making a CPU rather larger than today's small chips. The curved shape ensured that distances for cables could be kept short, which helped it achieve its impressive (for the time) computational speeds. This interconnect was used for communication between the modules (in the centre of the C-shaped machine) and the memory (located at each of the ends of the C), as well as between modules.

Modern machines have diverged from such designs, with interconnect between supercomputer nodes having become a completely separate component from that used within and between the CPUs, and between the CPUs and memory modules.

Currently, supercomputing centre managers may expect to spend around half the cost of the machine on its interconnect, and few would argue that it was not worth the investment. Although some systems are based on forms of Ethernet, many opt for the arguably higher performance of an interconnect designed for supercomputing. The most popular of these is InfiniBand (IB), although its more recent rival, Intel ®Omni-path (based on IP purchased from Cray) is rising in popularity now that it is included within the Intel processor package. In addition, there are custom interconnects used within purpose built machines (as opposed to those based on commodity components), such as Cray's Aries and Slingshot interconnects, as well as HP's (since they acquired SGI) NUMAlink, used for their Symmetric Multi-Processor (SMP) systems.

Despite their various differences, all of these interconnect designs attempt to keep bandwidth high and latencies low, so that the applications using them do not stall or become throttled by slow data transfer rates. As with memory access, a variety of tricks are employed by manufacturers to hide latencies, usually involving caching at various levels within the network.

Unfortunately, even the very highest specification interconnect has limitations. If many communication-heavy applications run simultaneously it is possible for the network to become saturated. Techniques employed to avoid this can only do so much, as eventually the hard limits of modern technology are reached, leaving the users of systems unhappy about the sudden slow speeds of execution [181]. Despite a lot of investment in research, there are no solutions to this problem at time of writing, and it is difficult to see how there could be without violating the laws of physics.

2.3 The power consumption/performance trade-off

Power has always been a problem in supercomputing. While the Cray-1 may have made it into several museums, the hardware required to power it has not. This machine also required its own rotary converter, producing electricity with a 400Hz cycle to smooth out the roughness of the incoming sine wave. This was coupled to various other pieces of power hardware, so more smoothing could be performed, before the power was finally clean enough to be fed into the fragile electronics within the huge C-shaped computer. This equipment had to be housed near to the Cray-1, and was generally placed in a separate plant room next door to the computer room.

Modern supercomputers similarly require a considerable amount of room to house their power infrastructure, though they do not usually need their own rotating converters (or generators) for normal operation. They are also very power hungry, and, while they may be more efficient (one can get many more FLOPS per Watt), they are also far bigger than the old systems, and so require more than the 115kW or so that the Cray-1 used.

Supercomputing centres are not the only customers to be crying out for greater power efficiencies. Computer usage is fundamental to modern life, and huge data centres are increasingly behind much of what we take for granted.

A result of the growing demand for power savings, and with the promise of such savings leading to greater chip sales figures, chip manufacturers have worked hard to reduce power consumption (and cooling, another big cost factor for data centre owners) of their chips. They have addressed the power usage problem in three especially significant ways:

- Voltage scaling
- Frequency scaling
- Powering on/off components

Both voltage and frequency scaling change the clock speed of the processor, and both were for a time controlled by separate electronics on computer mainboards. Sometimes this was configurable via BIOS settings. Now this hardware is (mostly) part of the chip itself, and may be controlled by microcode.

All three of these methods have performance implications, making them unpopular with supercomputer users. Some may even affect scientific results if detailed timing data is required. This is particularly noticeable in the case of powering off/on components of the chip, as floating point hardware is one of the parts to suffer from this innovation. The Floating Point Unit (FPU) and the vector hardware is powered down rapidly (if no instructions requiring this hardware are seen in the pipeline, and can take several hundred microseconds to power on again when required. For example, Skylake chips have been observed to temporarily use the lower 128-bit half of the vector execution units for 256-bit SIMD operations, to alleviate the problem of waiting the full 14μs for full-speed 256-bit SIMD instructions to become available [56]. Using the lower 128-bit half twice for a 256-bit instruction in this way is about 4-5 times slower, but means that overall throughput is higher. The units are powered down again if another 256-bit SIMD instruction is not seen again within 675μs.

A fourth design change in many modern chips is the elimination of 'dead' code by the CPU itself, rather than relying wholly on the compiler to do this. Increasingly, compiler optimizations are making their way into hardware too. The removal of redundant instructions is more than a mere time saving; each instruction has an energy cost associated with it. In

the case of chip manufacturers who target the embedded electronics industry, they have also designed their compilers to optimize for energy usage and executable/library size, rather than pure performance.

All of these changes have implications for HPC. Lower average power consumption means, for many data centres, that higher compute densities can be accommodated. During periods of lower power usage, the processors will also be producing less heat, so cooling demands are not as great either[9].

The end result of these and many other changes to reduce power consumption (and heat output) is that the speed of execution of the user's code is highly variable. "Turbo" can only be sustained for short periods, and it alone can account for as much as a 10% difference in speeds between individual cores; not all need be in "Turbo" mode at the same time, and their speeds differ slightly too [89].

Add to this the problem of chips being binned (an industry term for grading) by thermal design power (TDP), and the resulting variation between individual chips, and the amount of variation a user's code must now tolerate is very high. While the data centre managers may be happy with the situation, users of the machines must be able to tolerate the greater variation in processor clock speed and other effects, such as stalling or running some instructions more slowly.

Note that if a chip is being insufficiently cooled, its own internal thermal protection mechanisms will trigger, making the problem many times worse. Unless the problem is severe, no fault will be raised in the system logs, and the administrators may remain unaware while the users of their systems wonder what they are doing wrong.

2.4 Memory Hardware

In 1980, memory performance and CPU performance were very similar. They rapidly diverged, and despite many advances in memory design, memory has still been left far behind. Since most software applications are more likely to be memory-bound than compute-bound, memory access speeds have a marked impact on observed system performance.

This is especially true for scientific applications, the majority of which are designed for rapid processing of vast quantities of data. For these scientific programmers, it is the average speed of memory access (a combination of memory latency and bandwidth), not the CPU speed, which determines how quickly their software returns a publishable result. Since this is a difficult problem to address, many of the innovations in hardware, operating system and compiler design have sought to reduce the frequency of memory access, or at least to hide it when it cannot be avoided.

In addition to the ever widening gap in performance between the CPU and main memory, both have increased in complexity, and power saving features have been introduced that add additional latencies to many of the operations essential to numerical and scientific codes.

2.4.1 Virtual Memory

Hierarchical memory has been an expected feature of computers from the very beginning, and many early architectures split the memory in some way. This was not hidden from the programmer, and was a source of inconvenience and many bugs. Both the Ferranti Mercury

[9]Unfortunately, any supercomputer that sees high utilisation is unlikely to benefit from this, and it could be argued to be a waste of money to buy one that is not so well used

and the Atlas computer had two levels of memory: A small core memory and a larger drum memory. All transfers between the two had to be explicitly programmed, and were error prone. There was little in the way of protection, and not many were skilled enough to program these transfers efficiently, for example, by arranging accesses so that the drum memory need not rotate far, if at all.

Automatic management of these multiple levels of memory was not suggested (in print) until 1962 [106]. The Atlas had three levels of storage; core, drum (with an access time of 6μs) and tape, with the first two providing a large (for the time) amount of memory at a much lower price than a large core memory with the same capacity[10].

This was a commonly made compromise, but presented a problem for the programmer, who needed to keep track of where (in which memory) his/her program's data resided. It made a complicated program at least an order of magnitude more difficult to write, especially if any attempt at efficiency was to be made.

To address this problem, virtual memory was introduced. This is a simple concept, primarily implemented in the operating system, which presents to the programmer a single address space. In this way the multiple areas of storage appear as a single, large memory, even though the available working memory may be small.

In the implementation on the Atlas, Ferranti introduced a system that we now refer to as paging. The entire memory space is divided into equally sized blocks, known as pages. These pages may be moved between different levels of storage by the operating system, which may have limited assistance from the hardware, and which must keep track of such movements through an artefact known as the page table. This table keeps a record of virtual to physical address mappings for each page, and must be updated every time a page is moved. In the case of the Atlas, the natural size for these pages was 512 words, since this was the size of the fixed blocks on the drums and magnetic tapes. Through virtual addressing, the small, expensive, ferrite core memory, the drums and magnetic tapes could all appear as one large storage pool, with identical addressing; pages were transferred from the drum to the core memory by the Atlas Supervisor (described in more detail in section 2.9) automatically, reducing the cognitive load of the programmer.

Seemingly always with a focus on the trade-off between ease of use and performance, Burroughs had implemented a type of virtual memory through segmentation in their B5000 system in 1963. While this was not entirely transparent to the user/programmer, it was somewhat easier to work with than similar systems without any kind of virtual memory. It did not protect against memory fragmentation, so care was still required to use it efficiently. Interestingly this is a concept (as with stack-based processing) later used in the Intel 8086 processor, but which otherwise all but died out in the late 1970s.

However, it was a few years later (in 1972[11]) when IBM's 370 family of machines, which also heralded the first TLB, made the concept more widely available, implementing true virtual memory in the sense that a modern programmer might understand it. As today, the implementation was mainly a result of considerable work on the operating system (TSS on the IBM), making use of the hardware features designed to support it and rendering them invisible (to a greater or lesser extent) to the programmer.

The IBM 370 family implemented virtual memory with paging, as originally introduced on the Ferranti Atlas, rather than segmentation. Other hardware manufacturers implemented

[10] The Atlas actually had two small core memory stores, one for common routines, and one as working space for the Supervisor, and so inaccessible to the programmer.

[11] Although the IBM 370 family of machines began shipping in 1971, 370s with virtual memory did not ship until the following year

similar systems.

Multi-processing (also addressed by Burroughs in the B5000), complicates matters, as the memory regions of different processes must be kept separate, but the basic concept remains unchanged in modern computers.

2.4.2 Caches and Translation Look-Aside Buffers (TLBs)

Although for a short time, memory and CPUs ran at similar clock rates, getting the data (or the instructions) to the CPU in time to avoid it stalling has always been a problem. (For the purposes of this discussion, we will ignore caches that sit between main memory and non-volatile storage, and focus on those between the CPU and main memory.)

One way that was, and still is, used to hide the latencies involved in memory access is to use a small, faster memory near to the CPU and pre-fetch data (and instructions) to it before it is needed. This small, higher-speed memory is known as a cache, and acts as a buffer between the slow main memory and the CPU. The concept was first suggested in 1965 by Maurice Wilkes [179], who referred to it as a "slave memory", and neatly described a direct-mapped cache.

With its small size, the cache does not only bring higher speed (usually 5-10 times faster than the main memory on early machines), it also brings a different, though related, set of problems: Those of cache misses. A cache miss occurs when the data required by an instruction is not located within the cache. There are three main types of cache misses [81]:

Compulsory Caused by the first reference to a memory location; without very sophisticated prediction, this one is unavoidable.

Capacity Caused by a lack of free space in the cache; with its limited size, it is not possible for the cache to contain all the blocks needed for execution in all cases.

Conflict Perhaps more serious, and dependent on the placement strategy used for cache blocks (e.g. set associative, fully associative, or direct mapped), this is caused by two or more blocks mapping to the same position in the cache set.

Avoiding all three is of interest to both hardware designers and compiler writers, who must work together to achieve the best performance, despite their best efforts, they can only do so much to mitigate the problems. Alas, bad programming often leads to thrashing, which is when the same blocks are repeatedly evicted and reinstated in the cache, drastically reducing the performance of the running software.

The first commercial implementation was introduced by IBM with their 360/85 to avoid the need for large amounts of expensive memory. The TLB followed shortly after, made available in the IBM 360/67. This addressed some of the performance issues not addressed by the addition of a cache, by maintaining a table of recently used, and so translated, virtual addresses and their physical counterparts. In modern hardware, it is usually part of the CPU's memory management unit (MMU), and can be thought of as an address-translation cache.

2.4.3 Caches on modern CPUs

Modern CPUs achieve many of their staggering memory access speeds by careful use of several levels of caches, combined with both hardware and software prefetching to hide access latencies where possible.

The use of several caches was researched during the late 1980s, for example in [143], and found to be as good (in terms of performance) as using one large cache. This was good news for manufacturers, who could reduce costs slightly by avoiding buying large amounts of the very highest speed memory, instead using slightly differing speeds of memory for the different cache levels.

The level 1 cache, closest to the arithmetic logic units (ALUs), floating point units (FPUs) and registers, can be accessed much more rapidly than the lower level caches. On SMP or multi-core (the term used by Intel) chips, these caches may be shared by one or more processors (cores if using Intel's convention) residing within the same die or chip. Lower cache levels are usually larger than the higher levels, and data must be passed from one cache to another, up to the level 1 cache, to be operated on either directly or passed into registers.

Pre-fetching is employed heavily in both hardware and software (compilers often assist) to reduce the occurrence of cache misses. For programs accessing memory contiguously, the latencies involved in fetching data from main memory can be almost entirely hidden. Other memory access patterns are not so easily anticipated, and so application performance depends heavily on both the skills of the software developer and those of the compiler author. Both must have an understanding of the hardware to make choices in their designs that make best use of its features.

The use of multiple levels of cache introduces greater complexity, with memory management now shared between the hardware, the operating system and user software. Many performance problems occur when these three do not interact as the original designers expected, with thrashing (of either the cache or TLB) still one of the most common.

2.5 Symmetric Multi-Processor (SMP) systems

Burroughs produced a multiprocessor system, the aforementioned B5000, in 1961, but this was used in an asymmetric manner, with each processor assigned different tasks (one might run user programs while the other was used for the operating system). A user could not make use of multiple processors in their programs, but that would soon change.

In 1964, UNIVAC announced the 1108 II, which could have as many as three processors [114]. In the same year, another mainframe manufacturer, IBM, announced their dual-processor System/360. From that point onwards, the numbers of single processor mainframes available began to decline as customers demanded the ability to process ever larger datasets.

When the Cray-1 was unveiled, the speed of memory access had already dropped below the rate at which the CPU could process data, yet users of these and similar systems continued to demand the ability to process ever larger data sets. Cray Research's answer to this was to produce the Cray X-MP. Announced in 1982, this was a "cleaned up" version of the Cray-1, extended with two vector processors with shared registers, a large, shared, concurrently accessible memory and I/O subsystems. This design focussed on increasing throughput throughout, rather than simply adding more processors or increasing their speed. It had 8 times the memory bandwidth of the Cray-1, and the 32-way interleaved memory banks could be accessed independently and in parallel.

Users of the X-MP could run different programs on each processor, or one program that utilised both. This, coupled with binary compatibility with the Cray-1 (and the same COS operating system until 1986), probably contributed to the X-MP's popularity.

In the 1990s, it became possible to manufacture chips containing multiple processing units, with the first on-chip multiprocessor, a design by IBM, shipping in 2000. These on-chip mul-

tiprocessors, more commonly referred to now as multicore chips, saved power and time by bringing components into closer proximity. Shared last level caches, and shorter distances for snoop traffic, among other things, significantly improved performance for multi-threaded applications.

2.5.1 Simultaneous Multi-Threading (SMT)

Simultaneous Multi-Threading (SMT) was researched through the 1960s until it became widely commercially available with Intel's ®Northwood microprocessor in the early 2000's. Sun's UltraSPARC T1 and IBM's POWER5 followed within 2 years, providing the ability to run larger numbers of threads on the same processing unit at (approximately) the same time.

Several modern chips, with increasing numbers of processing units on the die, attempt savings in space and energy by sharing components. Although AMD was criticised for doing this with their Bulldozer architecture, where two processing units might share a single FPU, this approach seems to be proving increasingly attractive for chip manufacturers. It is less popular with scientific users, who have also struggled with increasingly Non-Uniform Memory Access (NUMA) architectures and rarely benefit from the use of SMT [128]. Unfortunately, scientific users are rarely the source of the majority of a chip manufacturer's income, and their pleas are likely to fall on deaf ears.

Although popular in mobile computing, heterogeneous processing units on a single die have not yet found success in supercomputing. AMD had some limited success in the desktop and server markets with their APU, a line of chips that include GPU components on the same die as the general purpose CPU. However, these chips were not popular with the computer gamers who provide the majority of AMD's income.

2.5.2 Non-Uniform Memory Access (NUMA) Architectures

As explained in in section 2.4.1, the system memory does not have to all be in one place, and the same is true when that memory is shared across multiple processors. In such distributed memory systems, it is rare that all processors can access every part of memory without incurring differing (time and latency) penalties. In these NUMA systems, the cost of accessing different areas in memory is dependent on not only the memory type, but its location. As virtual memory addressing gives the illusion of a single memory through a single address space, programmers must employ a variety of techniques to ensure that the data their programs are accessing is in a location that is not expensive to access, a cost that is dependent on where (which processor) the accessing process is running.

NUMA splits the system memory into regions, depending on cost of access. Programmers can make use of NUMA-aware libraries to determine which NUMA region their code is running in. On today's systems, NUMA regions may be "owned" by a particular socket.

2.5.3 Pipelining

Instruction pipelining, and pipeline chains, were one of the ways in which performance could be increased. Extensive use of these (and vector loops) was one of the ways in which the Cray-1 was able to out-pace its rivals. Through the use of pipelining, after an initial start-up time or latency period, a sufficiently well programmed CPU can produce an answer every clock cycle. Pipelines move operands through the functional units of a processor in a similar manner to a production line; one pair of operands enter a floating point unit or other functional unit as the

previous pair leave it to enter the next one. A sufficiently skilled programmer (often aided by a good compiler) ensures that gaps or "bubbles" do not occur in the pipeline, thereby avoiding a stall as one or more functional unit is starved of data.

By chaining pipelines together, the Cray-1 processor could achieve a high throughput.

2.6 Compilers

Although in theory the user can write machine code for a given machine, in practice this is very rare. Code is nearly always written in some high-level language, which can be compiled for a given machine, or indeed a set of machines, by appropriate compilers. In theory, the compiler knows about the machine it is compiling for, and it is the job of the compiler to produce the best code it can. In practice things are not so simple.

While some early computer designers, such as Robert (Bob) Barton who designed the Burroughs B5000 and its descendants in the 1950s, put a great deal of work into ensuring that the hardware was relatively easy to program[12], this did not last long.

As hardware became more complicated it fell to compiler writers to bridge the gap between the programmers and the hardware. Since the mid-late 1960s, compilers and high level languages have been essential to the delivery of almost all software projects, and the vast majority of developers could not make use of a computer without them. Also of note is how many are dependent on the diagnostic help of a compiler to find and correct simple errors, and to enforce language standards compliance. This support is now so ingrained that it is taken for granted, noticed only when it is absent.

In the Mythical Man Month [99], which chiefly deals with problems encountered with the increasingly popular IBM machines of the 1970s, an entire chapter is dedicated to comparing the development challenges and speeds of writing in machine code vs using a high level language. At that time, software was relatively simple, yet a five-fold increase in programming productivity is claimed. With the complexity of modern software, the gains are likely to be far higher. Even the early supercomputers were difficult to utilise effectively without the aid of a vectorising compiler. Porting software between machines has been made far less painful by the combination of high level languages and good compilers.

It could be argued that, as a society so heavily dependent on computers and the software they run, we owe a considerable debt to the writers of compilers.

2.6.1 Compilers in practice

Just as hardware has become more difficult for the end-user to understand, so too have compilers. Equally, few can make do without them. Indeed, a chip manufacturer could not sell a new chip without either providing compilers, or ensuring (typically by contributing effort) that the free open-source community is providing compilers. Some manufacturers do both. In addition to translating from a high level language to machine code (something that sounds far more simple than it is), compilers perform a number of optimizations at each stage of the compilation process.

Early compilers were monolithic in nature, but over time the compilation process has been broken up into several stages. The exact order varies from compiler to compiler, but in general

[12] The stack-based B5000, B5500 and B6000 used reverse Polish notation for expression evaluation, for example.

the first of these is performed by the lexical analyser (also referred to as the lexer or tokeniser), which breaks the program statements into lexemes or tokens for the parser to make sense of.

Very little optimization is performed at this stage, while a table of symbols is constructed which is eventually used to build an abstract syntax tree. Trees, being easier to reason about, are a good point for the more aggressive optimizations to begin to be employed.

Again, the order and nature of optimizations varies from one compiler to another. There is though a finite number of different changes that may be made to the code, and so the list is general. Optimizations are characterised as either machine-independent or machine-dependent, with the latter tending to occur during the later stages of compilation. Most of the former are applied to the abstract syntax tree (AST), intermediate representation, or machine independent code, with the intention of generating better performing machine code at the end of the compilation process.

Many optimizations are aimed at small code patterns that have a tendency to dominate program runtime, such as loops, while others rewrite code patterns identified as inefficient, such as branches. Branches are often straightened, avoiding unnecessary comparisons and allowing code that might not otherwise vectorize efficiently to be targeted for further optimization.

Loops may be manipulated in a variety of ways, not only to remove loop-invariants (i.e. code within the loop that does not change between iterations) but also to facilitate more efficient use of the hardware. Thus many loop optimizations, such as loop blocking, interchange or strip-mining, are performed to allow memory access to take place in a manner more suited to the design of the hardware. Compilers will also insert instructions to prefetch data into the CPU cache before it is required, thereby allowing data accesses to appear to happen more quickly.

Most compiler optimizations can be controlled, to an extent, by the user of the compiler, via a number of flags passed to the compiler at compile time. However, there are so many optimizations performed by the modern compiler that it is impractical to expect the compiler user to pick all the optimizations by hand. A long established practice among compiler authors is to define sets of optimizations or optimization levels that are chosen by the user of the compiler via the flag `-ON` where `N` may be a number from `0`, denoting no optimization, to `3` or `4`, the highest default level of optimization. Some compilers use `fast` instead of `4`. A further level, `s`, may be provided if the size of the resulting machine code is more important than execution speed, as may be the case in embedded systems.

This practice is so common and so well-established that it often leads to misunderstanding among users, who assume that, for example, `-O3` means the same thing to all compilers. However once the use cases of each compiler are considered, it becomes clear that the developers and maintainers of these compilers have different goals in mind and will attribute different meanings to these same flags.

Some compilers, especially some proprietary ones targeting a subset of machine architectures, may have best performance as their primary goal if that is what drives sales of their software (and hardware, in the case of chip manufacturers).

In contrast, a compiler used primarily by the open source software community to compile software for distribution in its binary (i.e. compiled) format is likely to take a more conservative approach to ensure portability between machines of several generations and manufacturers.

There are also the difficulties of managing user expectation. Code optimization takes time. Large software projects already run full build and testing cycles overnight or even, in some cases, over weekends, to try to ensure high productivity. If the compiler takes too long to compile a development team's software it will be very unpopular. On the other hand, they will also be upset if their software runs extremely slowly once built.

Compiler authors are required to find a good balance point between these two conflict-

ing requirements; that code compiles quickly and that optimization is performed aggressively enough to ensure that once compiled, the software runs quickly too. To give users control over this balance, and over what optimizations are performed, most of the many stages of optimization have user-callable flags that can be passed to the compiler to fine-tune the process, and in some cases to inform the compiler of the developer's priorities; sometimes smaller executables and libraries are more desirable than the highest performance.

Of course, optimization is only one of the useful things that compilers offer their users. As a critical part of the software development cycle, they also support a range of instrumentation and debugging options. Some of these may have an impact on the optimization passes, but this is not always stated in documentation. In contrast, the documentation of other compilers may be confusingly thorough, listing instrumentation flags under optimization because they have an impact on some part of the optimization process.

Some of the more obvious options are those that enable profiling of an application, as often the information gathered can be fed back to the compiler when the code is recompiled, guiding the optimization process.

2.6.2 Today's HPC Compilers

With the popularity of Linux, now the dominant HPC operating system, it is unsurprising that the most commonly used compilers are those provided by GNU. The GCC [154] supports a large number of different architectures and is heavily used, and supported, by the open source community. As a consequence, portability is an important consideration for GCC developers, with the result being that the compiler's pre-defined optimization levels (`-O2`, `-O3`, etc) are perhaps more conservative than they might otherwise be. On x86 systems, `-O3` only implied the generation of SSE4 instructions relatively recently, and still does not imply the generation of AVX instructions.

In addition to the various on/off flags, such as `-funroll-loops`, there are a number of parameters used to tune compiler optimization stages. One such example is `--param loop-block-tile-size=N`, which affects loop blocking or strip mining transformations, depending on whether the flags `-floop-block` and/or `-floop-strip-mine` are included. It defaults to a value of 51, which is curious; this was found to be referred to only in one entry on a GCC developer mailing list. The author described the change to 51 from 64 as being due to 51 being "more magic" [140]. This change took place shortly after the patch adding this parameter was accepted into the GCC source. It is difficult to see how the average user of GCC might be expected to guess what the most appropriate value might be for this (and similar) parameters.

A relative new-comer to this area is the Low Level Virtual Machine (LLVM) [160] compiler infrastructure. This provides C and Fortran compiler interfaces in the form of `clang` and `flang`. It is popular on ARM architectures due to support from Arm Ltd, and is rising in popularity elsewhere. Being around 20 years younger than GCC, its design draws heavily on lessons learned in software engineering. It is highly compartmentalised, with an emphasis on being easy to extend and maintain. Its development has been supported by Apple Inc for much of its existence, and it is used by other commercial companies as a basis for their own compilers and software development tools.

The compatibility of many of the `clang` command-line options with those of GCC highlights the dominance of the older, more monolithic compiler, but the flexibility and greater functionality offered by these LLVM-based compilers has lead to its use in a number of development-related tools [1]. Clang has also seen increasing support from open source projects

that could previously only be compiled reliably with GCC.

Currently, on the majority of architectures it does not rival GCC in performance and is not yet widely used, but that looks set to change as it grows in popularity with developers and hardware manufacturers. In scientific computing, one problem that has prevented wider use of `clang` and `flang` is that these compilers do not support all of the options available in GCC compiler for controlling optimisation, for example, it does not support `-fno-unroll-loops` or `-fno-move-loop-invariants`. For sensitive numerical codes, some optimisations, while they may make the code run much faster, cause it to produce invalid or inaccurate results. In such cases, being able to turn optimisations *off* as well as *on* is essential (as are good numerical tests).

Other open source compilers do not enjoy the same levels of popularity or attention as these two, and are not widely supported, so are not considered here.

In addition to the two popular open source compilers, there are a few well-used commercial compilers. One of the most well known, in part due to their domination of the x86 processor market, is that of Intel®. This compiler differs in approach to both GCC and LLVM, and the optimizations made when given `-O3` are less conservative.

The Intel®compiler suite generates code that uses CPU dispatching to choose from several versions of each function. The choice is determined based the output of the `CPUID` instruction to determine which Intel microarchitecture the executable is running on. If the decoded instruction output does not appear to describe a processor that the Intel compiler that generated that executable or library supports, it will fall back to a generic version that does not use any SIMD instructions newer than SSE3 [96].

This approach, and the more aggressive default optimization levels, ensure good performance on all of the supported (i.e. Intel-manufactured) processors, so long as a new enough version of the compiler is used to support any additional, performance-critical instructions. Unfortunately, some of the optimizations that correspond to the default optimization levels for this compiler are unsafe for scientific code, making it unpopular with developers of such software. This can lead them to recommend against using it, or to avoid supporting it, as the problems of numerical inaccuracy may be levelled against the software authors rather than the choices of the compiler writers. An example warning is given in the build documentation of the ATLAS BLAS library [174], which states:

> "Intel's `icc` was not tried for this release since it is a non-free compiler, but even worse, from the documentation `icc` does not seem to have any firm IEEE floating point compliance unless you want to run so slow that you could compute it by hand faster."

Whaley is thought to be exaggerating a little in this statement, but it is the author's experience that many (perhaps all) of the performance advantages of using this compiler come at the expense of precision, something that Intel admits to in [94]. For the majority of users of this compiler, that may be considered acceptable, but it is unlikely to be something that is (or should be) tolerated by many scientific users.

For those more focussed on supercomputing, Cray and IBM both provide their own compilers for use on their hardware [85] [37]. Since in both cases much of the hardware is their own proprietary design, and a close-kept trade secret, it makes sense to provide an optimized compiler for it.

2.7 Libraries

In practice similar problems are solved repeatedly, so it make sense to spend time and effort designing and implementing a suitable algorithm in a form that can be re-used. If this is sufficiently modular, and the design includes a well-defined Application Programming Interface (API), this could become a widely used standard. Such practices improve portability, regardless of whether the implementation is identical on all systems, because the various implementations of the API are indistinguishable from the point of view of the programmer using them. Such collections of frequently used algorithms are referred to as libraries, though not all are so successful in design as to be widely re-used.

In scientific computing, most tasks can be categorised into one of a set first identified by Phil Colella [34] and since expanded from the original 7 to 13 [10] [133]. One of these has an arguably greater impact on computer architecture than it perhaps deserves: Dense linear algebra. The reason for this is the TOP500 list [122], and in particular the benchmark it uses to rank supercomputers by achieved performance (R_{peak}); HPL [136].

This benchmark

> "solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers."

It includes precise timing and testing routines for checking the result is not invalid, but does not provide the dense linear algebra routines on which it depends. These are instead provided by a library, the BLAS [117], which as a result has been the focus of much work from chip manufacturers, library authors and compiler writers. All are intent on achieving a high score as positions in the upper reaches of the TOP500 are highly sought by their biggest customers, and help to cement a reputation as a supplier of the highest performing hardware/software. This has ensured that the BLAS, and in particular the `DGEMM` routine, especially as applied to square matrices, have garnered more attention than they might otherwise deserve.

2.7.1 Dense Linear Algebra

Of all the dense linear algebra libraries, the two most widely used and well known are probably Linear Algebra PACKage (LAPACK) [112], and the BLAS [117], on which it and the HPL benchmark depend.

The BLAS is an API for a collection of routines for the basic matrix and vector operations. These routines are split in to 3 levels depending on the type of operation being performed:

Level 1 Scalar-vector and vector-vector operations

Level 2 Matrix-vector operations

Level 3 Matrix-matrix operations (added in [43])

This well-established, clearly-defined API allows the software that depends on it to benefit from performance gains made by work to optimize its routines. Much of this work has focussed on double-precision GEMM for square matrices (`DGEMM`), driven by the HPL [136] benchmark and its role in the TOP500 [122] list.

Reference BLAS The reference BLAS is a naive implementation of the BLAS API, written in Fortran. The simplicity of its implementation makes it portable, as it is easily compiled

on any platform. However, as this implementation lacks any attempt at optimization (required to ensure clarity of the code), its performance (in terms of % of R_{peak}) is dictated by the optimization work of the compiler.

Many compilers will now spot the routine vital to HPL performance, `DGEMM`. This routine performs a matrix-matrix operation of the form:
$C = \alpha \times f(A) \times f(B) + \beta \times C$ where $f()$ is one of $f(X) = X$ or $f(X) = X^T$
α and β are scalars, and A, B and C are matrices. $f(A)$ is an m by k matrix, $f(B)$ a k by n matrix and C an m by n matrix.

In this reference implementation, although some basic checks are carried out to ensure a quick return if the computation is unnecessary, the basic algorithm loops through the elements of the matrices in a sequential fashion. For the form $C = f(A) \times f(B) + C$ where $f(X) = X$, the basic algorithm (ignoring simple checks to avoid unnecessary computation) is as follows:

Algorithm 1 Basic `DGEMM`

```
for j ∈ {1,...,n} do
  for l ∈ {1,...,k} do
    for i ∈ {1,...,m} do
      C_{i,j} ← C_{i,l} + (B_{l,j} × A_{i,l})
    end for
  end for
end for
```

Why other BLAS As can be seen in algorithm 1, one memory access is required per operation. While many optimizations are possible [167], these would defeat the point of a reference implementation by making it difficult for the reader to comprehend (and potentially less portable).

These memory accesses are optimized to some degree by the processor, since they access memory contiguously and thus benefit from prefetching, but this is not enough to deliver anything close to the floating point throughput that we know the processor is theoretically capable of (R_{peak}). In addition, this type of memory access pattern is severely affected by differences in hardware design.

GotoBLAS Kazushige Goto, a skilled hand-coder, is the sole developer of GotoBLAS, and the basic design is a fairly simple one. At compile time, the install script checks the processor type of the machine it is running on and selects a pre-written kernel from a small selection. As in [69], Goto himself states:

> Implementing all the algorithms discussed ... on a cross-section of architectures would be a formidable task.

GotoBLAS is thought of as 'portable' only in the sense that Goto has implemented kernels for a number of currently popular processors, although he also includes a generic (for x86) implementation.

Each kernel is written by hand in assembler, using various architectural quirks wherever they may convey an advantage. There are slightly different kernels for different matrix shapes and sizes, to ensure that not only square matrices are manipulated efficiently.

Unfortunately, development on GotoBLAS/GotoBLAS2 stopped in 2010 when Goto left his post at the University of Texas [183]. In 2011, development moved to OpenBLAS [180].

ATLAS The ATLAS (Automatically Tuned Linear Algebra) project [178] is an established project created and maintained by R. Clint Whaley. He explains in [169]:

> ATLAS is an implementation of a new style of high performance software production/maintenance called Automated Empirical Optimisation of Software (AEOS). In an AEOS-enabled library, many different ways of performing a given kernel operation are supplied, and timers are used to empirically determine which implementation is best for a given architectural platform.
> ...
> ATLAS is designed such that several kernel routines supply performance for the entire library. The entire Level 3 BLAS may be sped up by improving the one simple kernel, which we will refer to as gemmK, to distinguish it from a full GEMM routine. The Level 2 routines make similarly be sped up by providing GER and GEMV kernels (there are several of these, as discussed later). ATLAS has standard timers which can call user-programmed versions of these kernels, and automatically use them throughout the library when they are superior to the ATLAS-produced versions.

ATLAS is designed with maintainability [177] in mind, but that does mean that it makes some sacrifices in order to achieve it. For example, there are *no plans to have the code-generators produce machine-specific code* [169]. On the other hand, if you were to compile and tune ATLAS on a previously unknown architecture there is a reasonable chance that you would get good results. This is of course one of the biggest benefits of auto-tuning – the code is made more portable than hand-tuned alternatives.

There are also a number of proprietary BLAS libraries. Most of these are created by hardware vendors, who naturally wish their hardware to be compared favourably with that of their competitors, particularly in published benchmarks.

2.7.2 Fast Fourier Transform (FFT)

Though this thesis does not directly address the fast Fourier transform (FFT), these algorithms would be the next topic to be considered. Another of the Dwarves first identified in [34], FFT is a much more complicated algorithm than the BLAS, and is heavily used in scientific applications. It operates on complex numbers, which make it a difficult target for vectorisation, especially given the current lack of modern CPUs with complex SIMD instructions.

The most popular non-proprietary FFT library is FFTW [62]. This library attempts to self-optimize during runtime, with a limited amount of input supplied by the programmer calling the routines.

2.8 Software

The extensive variation in hardware has left its mark on some of the software standards that we rely on today, such as the C [25] and POSIX [142] standards. Many will be familiar with

the definitions of types, and particularly their sizes, in the C standard, though inexperienced programmers tend to forget that the standard merely sets their minimum sizes.

The original LINPACK benchmark [42] has a placeholder for the benchmarker to insert their own calls to a suitable timing routine, since there was no standard way of doing this at the time. In contrast, all one needs to run its successor, HPL [137], is to ensure that an implementation of the MPI and BLAS libraries, and a suitable C compiler, are available. All of these are governed by now-established standards. While all standards have failings[13], they still reduce the amount of ambiguity and help to make the lives of developers much easier.

At the time of the University of Southampton's 2014 experiments with 64-bit ARM, much software was missing, and a large proportion of what was available was buggy. Compilers were limited to Clang and GCC. While their support was good enough to get something to run, optimization was lagging behind as the compiler authors were using "generically good" options behind `-O3` et al. The high performance libraries were mostly absent. Of the BLAS libraries, only ATLAS was available and working, and that delivered a mere 20% of R_{peak} at the time. Without any tuning of the compiler flags, this was still far beyond what the reference BLAS could offer. It was here that some of the early prototypes for this project were tested.

Since then, Arm Ltd has produced more software for this area, with its own libraries and a compiler and tool suite based on LLVM.

2.9 Operating Systems

Early computers were supplied not only without compilers but without operating systems, something that is often forgotten by those who now depend on computers for their trade. They were single program, and so single user, who would supply the instructions either by punched card or paper tape. The earliest scheduling systems were entirely physical: Cambridge University famously used a clothes line, with punched paper tapes pegged to it, their order representing their positions in the queue, and the colour of the pegs indicating priority. Early accounting involved measuring used machine time by checking a clock mounted on a wall; modern workload managers still refer to time used as "walltime", or "wall-clock time".

Later, simple programs offering some of the functions now taken for granted in modern operating systems may have been supplied either by the manufacturer (on request) or by the customers, who would likely hire their own team of developers at the same time as, or shortly after, purchasing a machine. These people would have the laborious task of programming in machine code, and later, in assembly language[14].

Early operating systems were very rudimentary by today's standards, beginning as simple libraries and programs that ran before and after each user's program. As processing speed increased (along with the sometimes devious antics of users desperate to get as much out of their allocated time as possible), the complexity of these runtime libraries increased also, until they became a program that ran first, before any user processes, read in each user program, controlled its execution, performed any necessary cleaning up of the used resources and recorded data about resource usage, before moving on to consider the next user program. The clothes line had been replaced by software.

The Atlas Supervisor, famously introduced on the Atlas computer at Manchester in 1962, has been described as "the most significant breakthrough in the history of operating systems"

[13] Some might argue that the C and POSIX standards have considerable failings

[14] It could be argued that, since manufacturers supplied libraries and programs to translate from human readable symbols to machine code, compilers pre-date operating systems by a few years.

[73]. Its function was to manage the assorted components of the Atlas computer, with its many peripheral devices, including the different levels of memory. In this way it was not entirely unlike a modern operating system, though few modern users would recognise it as such.

The early operating systems also differed greatly from one manufacturer, and often machine, to another: The mechanisms for managing memory, processes, I/O and interrupts (if they existed) were unlikely to share many similarities, meaning that software would have to be rewritten, perhaps even redesigned, whenever a new machine was purchased.

The CDC 6600 was supplied with a copy of the Chippewa Operating System, which was really a simple scheduler or job control system. The first two Cray-1 machines to roll out of the factory were supplied without any operating system [38], but serial number 003 used the Cray Time Sharing System (CTSS), a descendent of the earlier system used on the CDC 6600. At this time it was already apparent that a large part of the cost of developing a supercomputer was in creating and maintaining the software. This is something that is still true today. It is therefore unsurprising that Cray, along with other manufacturers, has moved from their own, in-house operating system (Cray Operating System), choosing UNICOS (based on UNIX) for the Cray-2 (1985) and Cray X-MP from 1986, before eventually moving to Linux.

More recently, the TOP500 shows 100% of listed machines to be running some variant of the Linux (a UNIX-like) operating system. Such harmonisation has made working on software for these systems very much less surprising than it once was, and portability is relatively simple to achieve.

The POSIX standard [142], introduced in 1988, was an attempt by the IEEE to push operating system authors towards some sort of uniformity, and it has largely succeeded. Today, the most used variants of UNIX and UNIX-like operating systems are all POSIX-compliant, allowing many open source software applications to become widely used.

CHAPTER 3

RELATED WORK

Auto-tuning has been turned to by researchers for a number of years to try to find a way to improve performance without losing the advantages of portable code.

This project is targeted at the person, either a member of a research group or a system administrator, who has a piece of software (in source code, otherwise he has no control) that he wishes to run more efficiently than "out of the box" compilation provides. To make sense of this desire, he must first pick some "typical" jobs that he wishes to run faster, and attempt to improve the runtime of these. There are then two obvious options for doing so.

1. Rewrite it (presumably the expensive portions of it, an activity which in turn requires a non-trivial profiling operation to determine these) in machine code. This process is labour-intensive and requires skills that are in very short supply. Its results also have a short lifespan, as the next machine may completely invalidate this (if a different chip vendor is chosen) or at least will change the target.

2. Try a non-standard compilation. Obviously, if "out of the box" is -O0, one can try higher levels. Beyond this, there are many options, such as "should I change the value of `--param loop-block-tile-size=N`?". Doing this requires intimate knowledge of the compiler, and also some knowledge of the application: a combination of skills also in very short supply. This process is also liable to be labour-intensive and have the same short lifespan as the previous.

This project falls in the area of auto-tuning, packaging and automating option 2. It is the first in this area to combine a number of techniques (workload distribution, verification of compilation and statistical analysis of results, and portability of the tuner) and to target emerging HPC architectures. However, is it not the first time that libraries have been targeted to improve performance for a number of applications without those applications needing to be altered, or to attempt to do so through controlling compiler optimization by searching for a more optimal set of compiler flags.

While this approach is related to iterative compilation, it is different in that it does not attempt to integrate with a particular compiler, and a black-box approach is taken to the target software. It is not an extension to the compiler, but rather a method of driving it. This separation is maintained in order to remain compiler agnostic.

3.1 Auto-tuning

Auto-tuning is the process of taking a set of parameters and a target application or code, and searching for the best-performing variant of that application or code. Although "performance" may be defined in terms of energy or memory use, or some other indicator of how the hardware is used, in HPC most people are interested in speed of execution, particularly of floating point arithmetic.

Auto-tuning can be implemented at one or more of several stages in the life-cycle of the target software. It may start at compile or code generation time, or happen dynamically during the runtime of the program, or it may rely on data gathered during runtime to re-generate the code to improve the performance when the software is next used.

Choice of auto-tuning target also changes depending on the approach taken. Some software provides its own auto-tuning mechanism. This is the route taken by several library authors, of which ATLAS [178] and FFTW [62] are examples.

ATLAS employs auto-tuning during the build process to choose from a set of parameterised kernels, and to determine the correct parameters to use on a given machine. This auto-tuning takes place during build time, and relies on a long (perhaps 48 hours in some cases) period of uninterrupted time on a dedicated computer, identical (particularly in terms of the CPU) to that on which the software using the library will be run. A similar approach is taken in PHiPAC [16]. Both PHiPAC and ATLAS attempt a limited amount of tuning of the compiler flag choices during the build process. In contrast, FFTW tunes its kernels dynamically during the library's use (in addition to generating kernels at build time), with limited input from the user.

The work done on a particular library, no matter how good that work is or how widely used the library, can still only benefit a limited number of applications. To reach a wider, more general, audience, some have provided compiler extensions [64] or language extensions as in [103], [75] and [146] (often requiring bespoke code-preprocessors). The major benefit in this approach is that the developer can attempt to improve the reasoning of the compiler, thereby potentially allowing more aggressive optimisations to be performed. It is difficult for the non-developer to make use of these language extensions or annotations, as the person adding them must be intimately familiar with the application being modified. It may also be necessary to target "hot spots" after profiling the application, to save time.

The work in [64] is one of several (reviewed in [11]) that use ML techniques to improve compiler optimisation. This type of approach was considered early on by the author but discarded on the grounds that a lot of training data must be available first. This data must be available for the compiler and the architecture, which in practice requires a method of generating such data; a time-consuming process that is difficult to do portably. Milepost [64] itself is compiler and compiler-version specific, a trait that would not be suitable where portability of the tuner is important.

Where Milepost uses static and dynamic profiling to obtain information about the target application, [115] and [145] use hardware counters to make choices about compiler optimizations. While initially attractive, especially with the availability of Linux perf tools [22], hardware counters are unreliable: The author has several times noticed chips misreporting event counts, and recent specification updates for Intel processors have mentioned the problem of either not counting or counting events more than once (see for example CA93 in [89, page 42]). In the worst cases, counters have been observed to fail to count as many as $1/4$ of occurrences of an event, or count 4 times more occurrences than have actually taken place. In addition, hardware counters differ greatly between chips, and in bleeding edge hardware may be poorly

documented, as has been the author's experience.

The TACT project in [138] is much closer in goal and nature to this project, although it is not focussed on HPC. The authors use a genetic algorithm and a concurrent tuning approach to optimise software for several development boards containing ARM Cortex A9 chips. They state that it can take 1-2 days to get two-digit speed-ups for their target software. Such a delay is tolerable if a program is to run for a long period of time, as the cost in CPU-hours during tuning is regained through a shorter runtime. In scientific computing, that may mean more results or more detailed models, or it may mean a higher throughput for the supercomputer centre overall.

In [138], Plotnikov et al pass the compiler flag string (built from a subset of up to 200 flags) by setting the environment variable "`CFLAGS`". This is an obvious approach to take for open source software, where the convention of using `CFLAGS` to allow any person building that software to adjust the flags passed to the C compiler.

Some have created optimization frameworks for use with/on software that one does not wish to personally alter [12] [102] [8], removing the need to use a particular language to benefit from auto-tuning. These still require substantial work on the part of the user before they can be used. In the case of OpenTuner [8], it is implemented in Python, a language unsuitable (albeit popular) for HPC applications. The portability problems that this introduces are discussed in section 7.1.1.

Auto-tuning and benchmarks

Auto-tuning benchmarks can be a controversial topic, depending on the aims of the benchmarker. In [78, page 29], altering compiler flags for benchmarks attempting to compare the performance of different systems is strongly discouraged.

Hennessy's advice would appear to assume that the compiler will not detect the underlying hardware and make different adjustments dependent on what it finds. In these times of clever compilers, a true comparison of the hardware might require that the benchmarker employs no optimization at all. Even then, the hardware itself may make benchmarking difficult, as seen by Fogg when attempting to time instruction execution in modern chips [56].

One benchmark that has perhaps affected compilers (and hardware designs) is the HPL benchmark [136]. As the `DGEMM` algorithm is relatively trivial for the compiler to discover, it is an obvious target for aggressive optimization. It might make for a more interesting TOP500 [122] if both unoptimized and optimized versions of the benchmark (and the BLAS library on which it chiefly depends) were run, but this is unlikely to be popular with compiler or hardware vendors, or the benchmarkers who are chiefly employed to guarantee as high a result as possible is obtained from the benchmark.

3.2 Tuning the choice of compiler flags

While many are aware of and have discussed the way that ATLAS chooses kernels and suitable parameters, they may not have noticed a C program called `mmflagsearch.c`, mentioned only briefly in the install documentation [171]. This simple program uses a greedy linear search to find the best combination of a small set of flags provided in a text file. It does not search values for specific flags (e.g. `--param loop-block-tile-size=N`), but rather tests whether the flag has a performance impact or not. For flags with multiple values, all possible valid values for those flags must be listed, making the approach impractical for large

numbers of flags that take a range of values. It also does not consider dependencies between flags, although there appear to be very few of these, nor does it attempt to evaluate the benefits of particular combinations of flags.

The list is hard-coded and began with only a small handful of flags, hinting at its origins as a small experiment. The list has grown as more have been found by Whaley, the ATLAS author, to produce good results generally when compiling a GEMM kernel. Only machine-independent optimizations are considered. The `mmflagsearch.c` program can only be used with GCC, but this is also the only compiler that is stated as being (fully) supported for use with ATLAS.

Whaley's experience mirrors what others (such as [131], [84] and [141]) have found: That careful choice of the flags provided to the compiler can out-perform the choices of the compiler writers that optimization levels such as `-O3` represent. This does not need to be automated if the individual making the selection is knowledgeable enough about both the compiler and the code being compiled, but such knowledge is seemingly rare.

In [131], the authors focus on eliminating flags from the search, a useful technique in a large search space, but only consider the small subset of flags corresponding to `-O3`. Similarly, in [84], the 60 on/off flags corresponding to `-O3` of GCC at that time is searched for the best combination. Cole et al report an improvement of 53% over the standard optimization levels. They consider only the SPEC benchmarks [155], and the authors do not concern themselves with the possibility of numerically unsafe optimizations being performed.

Closer to this project is the example presented in [8], which tunes all the flags and parameters of GCC via a simple genetic search algorithm. This choice of search algorithm is similar to that employed in [6], in which a very similar (in nature) search space is described. However, Almagor et al were able to explore their much smaller search space, and some of its subspaces, exhaustively. This is not feasible with GCC, or any of the popular modern compilers.

[110] and [35] address a similar problem, also searching for the best ordering of optimizations in a bespoke adaptive compiler. In the project in this thesis, targeting an existing compiler, the only option is to turn the optimizations on or off, adjusting them where possible via one of the provided parameters. No widely-used compiler allows for the order of optimization passes to be tuned by the user.

Of perhaps more relevance, in [130] GCC flags are searched for a "good" combination, this time tuning with the aim of reducing energy usage.

In [141], Popov et al use CERE [40], a codelet extractor built on LLVM [160]. Their tool targets the generated bytecode to tune further compiler optimizations of small sections of the targeted code. This made possible by LLVM's use of bytecode and modular design. Until LLVM becomes more widely used, the potential for this approach is limited to those codes that support it, although this list is growing.

[166] tunes the optimizations of hot code segments by the Intel ®compiler on Itanium, also targeting some of the SPEC benchmarks. Again, a greater than 20% performance improvement is reported, which would be impressive if it can be repeated with real codes, without unwanted side-effects.

3.2.1 A note about optimization levels of compilers

As explained in section 2.6.2, the optimizations corresponding to `-O3` and other optimization levels are picked by the compiler authors, who have a specific use case in mind.

It should also be remembered that `-O3` is revised by the authors of the various compilers when work is done and changes made to the optimization stages of compiler. If the stated

goal of a tuner's author is to out-perform `-O3`, then it would be foolish to imagine that this is anything other than a moving target.

This is not to say that compiler writers do not know what they are doing when these choices are made. On the contrary, it highlights two problems: Choice of flags is dependent on both the architecture (and microarchitecture) of the target platform and on the code being compiled. For the GCC authors, their choices for `-O3` must be made with the understanding that the target platforms are diverse, as is the software being compiled. Portability is an important consideration. It is also difficult, as stated in [131] for the compiler to know enough at compile time (without taking longer than is likely to be tolerated by the developer) to make optimal choices.

For some proprietary compiler authors, such as Intel®, the problem is slightly easier. They need only target their own hardware, and, unlike the GCC developers, have no need to worry about Linux distributions using their compiler to create binary packages for installation across every hardware platform that distribution supports (and some it does not).

Even when the authors of compilers are, as one might imagine is the case for those employed by Intel, intimately familiar with the hardware, they can never claim such familiarity with the software being compiled. They cannot know what it will be, or what it will look like, and so their choices for `-O2`, `-O3` et al must be based on a great many assumptions about what a "typical" application looks like, and how it will behave.[1]

A relevant example is found in [105], where, for the specific problem being investigated with certain combinations of input sizes, `-O1` may actually yield better performance than `-O3`. A 20% improvement in this instance, contrary to what might otherwise be expected.

3.3 Distributed compilation

Over the past decade, distributed compilation has increased in popularity. This would appear to be due to the huge growth in large software projects and demand for tools to manage them, accompanied by a drop in cost of commodity hardware. Although several distributed build systems exist, none are intended for use in the HPC environment, and benchmarking is not a normal part of the software build cycle[2]. They also do not concern themselves with jitter [39], which we know, for example from [119], can have a noticeable impact on benchmark results.

The many features that make distributed build systems and compilers so interesting to software developers could add considerably to existing machine noise. There is little incentive for the maintainers of these projects to pay attention to this, since the average software project, their main use case, makes little use of auto-tuning and so is not adversely affected.

Popular examples are DistCC [139], expanded by others, such as the SUSE team [157], to create their own custom distributed build systems, and MRCC [116]. DistCC is written in C++, and is really a wrapper for GCC. It requires a daemon running on dedicated build servers, and does not allow for benchmarking or testing of optimization results. SUSE's "Icecream" (their customised version of DistCC) also requires a daemon to be running on the build servers, as well as at least one server for distributing the work. Koji [108], a distributed build system used by the Fedora Linux project [54], works in a similar way, but its goal is to build RPM packages for distribution (after signing via another distributed system). This means that each build task

[1] To claim otherwise would suggest the possession of a level of psychic ability that surely would grant the possessor the prospect of a far more lucrative career than that of a mere software developer.

[2] Though companies often do benchmark their software, it is generally done manually and near the end of the release cycle, if at all.

will run a variety of scripts, one of which usually includes a "`make test`" if available.

MRCC is designed to compile large software projects on MapReduce servers, so in some ways it fits better with the HPC model. Compilation jobs are submitted to the MapReduce scheduler, and no dedicated nodes are required. It too does not allow for benchmarking or checking whether optimization has caused instabilities, but in theory these tasks could be created as stand-alone MapReduce jobs and run in a similar way.

A more interesting project published at the end of 2017 is the work of Mohamed Boussa [21]. This project uses Docker [41] containers to get around many of the difficulties in using distributed build systems on a university cluster. Unfortunately, Docker is not without its own difficulties, and discussions around its appropriateness in that environment tend to provoke heated debates. This is particularly true among those responsible for running and maintaining university HPC facilities. Despite this, it is very popular with users, and so the US Department of Energy has produced two projects, Singularity [111] and Shifter [126] to allow Docker images to be ported to a format with lower performance overheads and slightly less noise. Both seem to work in a similar way. This involves unpacking the Docker containers into a `chroot`, and placing this within a more space-friendly `squashfs` image. This also helps to placate those systems administrators concerned about the proliferation of Docker exploits, although it cannot alleviate all their fears.

CHAPTER 4

INITIAL EXPERIENCES

> An unhealthy preoccupation with floating-point arithmetic's Speed, as if it were the same as Throughput, has distracted the computing industry and its marketplace from other important qualities that computers' arithmetic hardware and software should possess too, qualities like
>
> Accuracy, Reliability, Ease of Use, Adaptability, ...
>
> William Kahan [100]

When a new supercomputer is bought, a great deal of money is spent on obtaining the latest and greatest hardware. While there are good reasons for this, it does present a problem during the first 1-2 years. Much of the software on which scientific applications rely requires time and effort on the part of the maintainers of these applications, and in particular the library authors, to optimize for the latest architectures.

Similarly, the administrators of these machines wish to be able to provide tuned software as early as possible, since shorter run times make for higher throughput. In addition, there are financial arguments for production systems. The placement of a system within the TOP500 [122] list may be important to many, but it is rarely justification for such investment. Why pay so much for the latest hardware if your users' applications do not perform as efficiently as on older hardware, albeit that their jobs may run faster than on the previous system? Research supercomputing centres must balance the needs of those who need access to bleeding edge hardware with those who just need a good number cruncher. Furthermore, it could be argued that the more modern the hardware at time of commission, the longer the useful lifetime of the machine.

Gaining access to hardware in a timely fashion is often a problem for maintainers of open source libraries. Even those working on closed source libraries, including those in a relatively privileged position such as the developers of Intel's®MKL [88], require time to discover and implement the best algorithm variants for that hardware.

When the first Knights Corner (Intel Xeon Phi) cards became available in 2012/2013, many invested, but they were let down primarily by the libraries. The author, working on the `Iridis 4` (appendix A.2.3) cluster at the University of Southampton, was disappointed to discover that very little effort had gone into the MKL at that time. Only `DGEMM` operating on square matrices appeared to have reasonable performance, and this remained the case for at least the next 2 years. Open source libraries were even slower to deliver, and maintainers on mailing lists were

seen stating that they were struggling to obtain access to the hardware and so could not support it [172].

The period of time between a supercomputer first entering production and high performance implementations of the libraries used by the software that runs on them becoming available is observed to be between 1 and 2 years at the current time, and appears to be lengthening as CPU architectures become more complex.

This increase in complexity can be seen by looking at the increasing numbers of CPU errata issued by chip manufacturers. Between the first specification date for the Xeon E7 v2 (IvyBridge) processors issued by Intel in November 2013, and the version issued in September 2017, the number of errata swelled from 106 to 162 [90]. In a similar time period (September 2014-2017), that for the Xeon E5 v3 family (Haswell) swelled from 76 to 119 [91]. The errata in the specification update for Skylake, first issued in July 2017, already number 71 [93], and are expected to continue to increase.

It has long been an annoyance that the manuals of the chip manufacturers may be ambiguous or otherwise diverge in subtle ways from the observed behaviour of the hardware. Such divergence has been noted by both library authors and optimizers [57] [58].

Even experts struggle to understand chip designs fully. In [98], it was shown that the well known and respected author of GotoBLAS [68] seemed to have mis-described the importance of his treatment of the DTLB.

4.1 Case study: When cache matters, a comparison of ATLAS and GotoBLAS

In early 2006, the University of Bath took delivery of its first modern supercomputer, `Aquila` (appendix A.2.1), a cluster built from off the shelf components, typical of those installed at many universities in the UK, EU and US. The first step after delivery was to put the system through a series of rigorous acceptance tests, ensuring that what was delivered was fit for purpose and met the requirements of the sales contract. The author, being at that time one of the two systems administrators responsible for HPC systems at Bath at that time, was heavily involved in this process.

During this stage, the engineer sent by the vendor to install the system ran into problems. The system was contractually required to achieve an observed HPL benchmark performance (R_{max}) of 80% of its theoretical peak (R_{peak}), but nothing that the engineer did would allow this figure to be reached. Eventually, the BLAS library being used with the benchmark was switched with another, and the system managed to pass this test. Most did not ask after the cause, but some, including the author, were left wondering.

4.1.1 Comparing BLAS library performance

Work began by looking at the difference in HPL benchmark performance on `Aquila` (see appendix A.2.1) using the GotoBLAS [67] and ATLAS [178] BLAS libraries. There were two versions of GotoBLAS on `Aquila` at the time, and these were compared first, as it would take some time to compile ATLAS 3.8.2 - at the time the latest stable version. This first comparison between two versions of the same BLAS library yielded surprising results.

The processor installed in `Aquila`, as explained in appendix A.2.1 on page 121, was an Intel chip, the microarchitecture of which was given the development name of Penryn. It was not long since this chip had been released, as is often the case for machines of this type and

for supercomputers in general. The older version of the library delivered lower performance, around 20% slower than the later version. This was initially blamed on support for the processor being lacking, but digging into the source of the library build system suggested a more interesting reason.

It turned out that the library's build system, largely driven by custom scripting of the developer, had misidentified the processor. The library produced was in fact optimized for an older Intel microarchitecture, Prescott, and as a result was unable to reach the level of performance usually expected for this library.

It later became clear that, during the work that he did to fix the performance issue, the engineer working on the system at the time made contact with Goto, author of the GotoBLAS library, who had needed access to `Aquila` in order to fix his software. With the library building correctly, using the kernel hand-crafted for the Penryn microarchitecture, the HPL benchmark easily managed 78.8% of R_{peak}[1]. This was close enough to the required 80% to allow the system to pass that part of the acceptance testing.

With that complete, attention turned to the ATLAS BLAS library. This library is very different from GotoBLAS. Whereas the GotoBLAS build system picks from a small set of hand-written kernels, with a few parameters set on a per-microarchitecture basis, the ATLAS build system is considerably more complex. Goto requires access to the hardware his library supports. In contrast, ATLAS auto-tunes, searching through a number of parameterised, relatively generic kernels in an attempt to provide a high performance library even for architectures (and microarchitectures) that Whaley, the ATLAS author, has no knowledge or experience of.

In practice, this approach is not always able to deliver such high performance as might be hoped, but it does manage to produce something far better than the reference BLAS compiled with `-O3` with any of the established compilers[2]. In the case of `Aquila`, this version of ATLAS was able to deliver 55% of R_{peak}.

Later versions, after significant development, are capable of achieving far more, though only on x86 architectures. Other architectures are significantly different that such good performance is difficult to achieve even for ATLAS.

4.1.2 Observations of TOP500 list

An analysis of the November 2008 TOP500 [122] list shows that, if reported R_{max} is considered as a percentage of R_{peak} for each machine listed, a bimodal distribution becomes apparent. The peaks are observed to correspond approximately to the differences in performance between ATLAS and GotoBLAS on `Aquila`. Unfortunately, TOP500 does not state which BLAS library is used with the benchmark (nor which compiler or combination of compiler flags) to obtain the submitted figures, so it is not practical to determine whether this distribution is due, either wholly or partly, to choice of BLAS library.

4.1.3 Comparing ATLAS and GotoBLAS

Why was there such a big difference in performance between these two, both impressive, BLAS libraries?

[1] This increased to 85% when the benchmark itself was later rebuilt by the author with a more careful choice of compiler flags

[2] Note that for the machine in appendix A.2.4, ATLAS at that time could only manage 20% of R_{peak}; still significantly better than the 2-3% of `-O3`.

In [67], Goto claimed that the superior performance of his library are due largely to his approach to the (presumably data) TLB, chiefly work on avoiding TLB misses, observing that:

> for the current generation of architectures, much of the overhead comes from TLB table misses

and goes on to state that the GotoBLAS approach is really focused on avoiding these misses, additionally informing the reader that:

> prefetching can mask a cache miss, but not a TLB miss

this being because a TLB miss stalls the CPU while it waits for the TLB to be updated. The GotoBLAS library is then compared with the performance of ATLAS (and Intel's MKL) on one machine, the Pentium-4 based production system at UTACC. Approximately the same gap in performance on his multiprocessor, Intel Pentium 4, machine was also observed on `Aquila`.

Interestingly, while the difference in performance could be reproduced on both `Aquila` and a machine with a very similar chip to that used by Goto, insights provided by profiling did not seem to agree with Goto's assertion about the TLB. ATLAS did not see more TLB misses than GotoBLAS when used with HPL on these systems and, further to this, no other thrashing was observed.

In support of his argument, in [67], Goto also quotes results for small matrices of only 100x100 elements, where high numbers of TLB misses are unlikely even for naive code, GotoBLAS still significantly outperforms its rivals.

Despite his claims in [67], it would appear that it is actually Goto's superior assembly that is the likely cause for his performance gains. ATLAS only has a large number of TLB misses when it deems the matrices too small to be worth copying into cache. This is discussed in the ATLAS author's own paper on the subject [178], where it is stated that unless the matrices being operated on are above a certain size (this dependent on architecture and found empirically by ATLAS at compile time), it is more expensive to copy the matrices into cache than to operate on them where they are in memory, rather than to copy and transpose. Once it begins using the implementation that does copy and transpose the matrices then you see very little difference in performance, regardless of matrix size.

Through profiling, it becomes clear that with these versions of the BLAS library, there is a big difference in how they use the processor cache. While discussing the matter with the ATLAS author, Whaley explained [170] that he also believed that the difference was largely due to treatment of the cache. In later versions of ATLAS, this was fixed by allowing the ATLAS build system to consider blocking for any level of the cache [173], which seems to be the more sensible approach, given the increasingly complicated memory arrangements of modern processors.

Prior to this, ATLAS would block only for the L1 cache, whereas GotoBLAS would block for the L2 cache, something that Goto mentions in [69] (where, perhaps tellingly, there is no longer any emphasis given to treatment of the TLB). Since the changes to the ATLAS build system were made, the performance gap has narrowed significantly, so that on many architectures it matters little which library is used.

What the work of both Goto and Whaley has highlighted is that, when last level cache (LLC) bandwidth is sufficient - as it apparently is on current machines - then there is no real advantage to be gained in blocking for the L1 cache. Thus if the block size is NB, then you can dominate the NB^2 load costs with your NB^3 flops if you pick NB to fit into the $min(DTLB, LLC)$ instead of L1.

The bandwidth to the L2 cache (L3 and possibly L4 in later chips) is generally more limited than to the L1 cache, so it is necessary to copy to special formats that have a fairly high spatial locality (according to Whaley [173], greater than those used in the ATLAS block-major format in 2014) so that you can be certain that you do not exceed the L2 bandwidth. This was a problem on many earlier chips, which is why Whaley, in focussing on portability over performance, continued to block for the L1 in ATLAS despite knowing of these advances on newer x86 chips for a few years. It also required considerable work to ensure that portability was maintained and future architectures would not be harmed by moving to a different blocking strategy. GotoBLAS does not support many chip architectures, and so enjoys an advantage over ATLAS where ease of maintenance and speed of development is concerned.

Neither ATLAS nor GotoBLAS is novel in the way that blocking is done to suite the available data caches on the chip being used. A similar approach was taken almost 10 years earlier in [79], and is also used in several proprietary libraries. In [143], simulations were used to show that several smaller caches could perform as well as one large one, and, although Przybylski did not have the BLAS in mind, it is not a great leap of imagination to see how this work might apply to those writing such libraries.

4.1.4 How much does the TLB really matter (to GotoBLAS)?

Intrigued by Goto's claims about data translation look-aside buffer (DTLB) treatment in [67], a short study was undertaken by the author [98]. This made extensive use of variables in the GotoBLAS build system that related to the DTLB, in particular its size. It became clear through examining the performance counters on the Penryn chips of `Aquila` that DTLB misses were not affected by changing the reported size of the DTLB where it was used by the build system. Further examination of the source seemed to suggest that in fact, changing the reported size of the DTLB affected very little in the code, which is supported by there being almost no practical affect on the performance of HPL. The main conclusion that can be drawn from this is that GotoBLAS is even less portable than it first appears, and also more difficult to maintain, as most of the important variables (along with several architecture-specific performance "tricks") are hard-coded in assembler.

4.2 Towards a deeper understanding

There are several guides on the internet about how to choose the best numbers for running HPL on your own computer in order to find the highest R_{max}. In most cases, the authors tend to quote from "the Quick and Dirty Linpack Benchmark Howto", a resource that seems to have since disappeared from the internet, or to give numbers that they have obtained experimentally.

The numbers that they find could have been estimated had the authors had an intimate knowledge of how the HPL and BLAS algorithms work, along with some facts about their hardware.

In particular, the `DGEMM` implementation in the BLAS library that they pick will make a big difference to the best numbers chosen for use with HPL. In the example at the beginning of this section, because that version of ATLAS blocks for the L1 cache, the number 80 (or 160 for some of the more modern chips with larger caches) gives the best performance when chosen for NB, the block size. In contrast, because GotoBLAS is blocking for L2, then depending on the size of your L2 cache you might choose 112 or 224 for the same processor. On `Aquila` in 2009, the best NB turned out to be 160 if using HPL with ATLAS and 224 if using GotoBLAS.

A small NB is good for communication purposes, but for performance it is still necessary to pick the largest size that will fit into the data cache size that the chosen BLAS library is blocking for. This is because of the sizes of the caches most used by those GEMM algorithms, and the way they are treated by the GEMM.

This does not account for the problems caused by thrashing, for example if data is cache mis-aligned or if threads are fighting over the same pages in memory, but it is hoped that most users of HPC will avoid these issues by having access to well-written, high performance libraries.

4.3 Early motivation

In 2012, the RaspberryPi [61] helped to increase interest in the ARM architecture among those outside of the mobile and integrated worlds it has previously tended to inhabit. This resulted in a great deal more support for the architecture among open source projects, though the focus of the majority of the community has been on desktop and more general purpose server software. Nevertheless, it helped to spark serious interest in the world of HPC, as the back of the work required to create a scientific computing ecosystem had already been broken.

When the University of Southampton unveiled its RaspberryPi cluster [36], it was not really expected that it would be a machine that performed at anything like the speeds seen from the much higher powered x86_64-based supercomputers in the university's data centres. Only two years later, a more promising ARM-based cluster arrived, populated with APM X-Gene 1 processors. This also could not compete, its vector instruction set being limited to the approximate equivalent of only MMX, but it showed how quickly the gap was closing and how much work was still to be done.

At time of writing (early 2018), no TOP500 systems are ARM-based, and more than 95% of the listed CPU architectures are x86[3], but the author believes that it is only a matter of time before a more balanced mixture of CPU architectures are seen in the list again[4].

[3]In both the June 2017 and November 2018 lists, non-x86 chips make up just 4.6% and 4.8% of each list respectively, with only 3 of the top 10 systems in November being non-x86.

[4]Returning to this statement in March 2019, the most recent top 10 list contains only 6 x86-based systems, none of them in the top 3.

CHAPTER 5

REQUIREMENTS ANALYSIS

Having decided on an area of approach, that of auto-tuning the selection of compiler flags, further work was necessary to fully understand how this might be manifested in a useable tool. Before and during design of this tuning tool, as well as during its testing, some investigation would be required.

Not only would several prototypes be necessary to develop a detailed understanding of the problem area, and inform the final design, but the platforms on which the tests would be run had to be benchmarked (sometimes referred to as baselining in this context). The benchmarking activity would provide data on the baseline performance of the target software on the target hardware, under target conditions. This last activity was necessary both for comparing the tuning results to the baseline performance figures obtained from using the compiler's built-in optimization levels (`-O2`, `-O3` et al), and to obtain correct input values to the tuner itself.

Throwaway prototyping was used to give a better understanding of the problem area. Initial prototypes were written in Python, for speed of development, with the prototype 5 making use of the OpenTuner [8] framework, though this proved to be unwieldy.

As expected, the early prototypes required discarding and re-designing frequently. Despite the issues with portability, discussed later in section 7.1.1, Python, and the framework provided by OpenTuner, were useful for this early stage. They increased the speed of development greatly and allowed the author to gain a better understanding of the task at hand in a shorter amount of time. Ansel was also able to provide assistance, in the form of advice and a simple example, for which the author is grateful.

5.1 Initial High Level Use Case

In order to begin prototyping, it is necessary to have some idea of what the problem being addressed is, with respect to the proposed solution. At the beginning of the work, despite the potential broad application for the resulting tool, the decision was made to target libraries for HPC. This was judged to be of the greatest value to the academic community, potentially increasing significantly the throughput of a research supercomputer early in its production life.

The task could now be restated thus:

> Create a tool for auto-tuning the choice of optimization flags of a compiler during the building of a library for users of HPC systems, with the aim of producing a higher performing version of that library, on the target hardware, than might be obtained without such assistance.

There are three additional goals for the project implied in this statement, which would be further defined through knowledge gained during the prototyping process, and through experimentation. These are:

(i) The tool shall work on as many supercomputers as possible

(ii) The tool shall target at least the most commonly used/available compiler

(iii) The tool shall be flexible enough to target the most commonly used scientific libraries

Goal (i) is best answered by looking carefully at the TOP500 [122], and in particular at the operating systems used in those performance figures. While TOP500 is perhaps not the most inclusive list, consisting only of those machines whose owners wished to publish details, it is thought to be representative of the supercomputing landscape. The more eccentric designs tend to be found around the top of the list, but list is dominated by the more commonly used (and cheaper) clusters. The common features of these machines are broken into slightly more detail in list 5.2.

Goals (ii) and (iii) are hopefully obvious to the reader. In order to be portable, the ideal tuner would be flexible enough to work with any compiler. Fortunately for this project, most supercomputers now use some form of Linux, with a very few outliers still covered by POSIX-compliant UNIX of some kind. Operating system portability is therefore relatively simple to ensure.

Likewise, in the compiler space most systems will have GCC available. Although Clang (from LLVM [160]) is becoming more common, the dominance of GCC means that Clang behaves similarly in terms of what options it takes on the command line in order to maintain compatibility.

Targeting libraries, as stated in requirement (iii), seems an obvious choice. It could be considered that libraries are already the target of many systems administrators, who take the trouble to install high performance versions, confident that most scientific applications will make use of these [34]. While it is possible to narrow down the most commonly used routines within individual libraries and target those separately, as has been done in approaches such as [40], this type of approach is unable to keep up with the rapidly changing world of scientific software. A library that has only some routines tuned is liable to require re-building when the un-tuned routines become more heavily used in future.

5.2 Requirements Specification

At the beginning of the project, having looked at the current state of software in high performance computing and the problem previously described, it was not too difficult to draw up a preliminary set of requirements with which to begin work on the early prototypes.

To begin with, some simple assumptions were made:

- The environment would be that typical of most HPC systems:

  - Linux OS
  - x86_64
  - Several homogeneous nodes
  - No virtualisation
  - Dedicated access to the compute nodes doing the tuning (ie. not shared with other users)

- The person using this would be a systems administrator or application support specialist, responsible for building software for use by those working on a supercomputer
- Any benchmark used tuner will be run for sufficiently long to represent user code
- Any benchmark used will give consistent results (similar run times) when run repeatedly under the same conditions

These are based on typical HPC facilities, such as those at UK academic institutions. The TOP500 list [122] has shown towards the lower end of the rankings that these sorts of facilities are the most common. It also helpfully lists operating systems as well as hardware. For the duration of this project, Linux adoption was above 90%. By the end, it had reached 100%, although there is some discussion about whether this is a true reflection of the top end machines[1].

5.2.1 Initial Requirements

The design was largely use-case driven (using the use-case implied by the task stated in section 5.1), with additional non-functional requirements.

Non-functional Requirements

1. Shall be portable between different machine architectures
2. Shall be able to target a variety of codes[2]
3. Shall be able to target at least the GCC compiler
   (a) Ideally, should be able to target other compilers
4. Shall work on a single host or on several
5. Shall work on Linux[3]
   (a) Should also run on AIX, BSD, other *nixes
6. Should not require a steep learning curve or specialist knowledge on the part of the user

[1] It was pointed out to the author by colleagues at a DoE lab that the high end machines, such as those provided by IBM and Cray, frequently use one OS for the login/front end nodes, and something custom on the compute nodes. This is not reflected in the TOP500 list statistics.

[2] at least one of the Dwarves [133] [34]

[3] The TOP500 list in July 2016 showed 96% of the publicly known about supercomputers run the Linux operating system. By the following year, the number had risen to 100%.

These requirements were all taken into account during the creation of the first prototype, but some were ignored or simplified in order to explore the problem space more rapidly. The early prototypes only considered GCC, and in fact the data about GCC was hard-coded in the first and second one, so only one version could be targeted initially. This hard-coding was mostly absent in the later prototypes, but it was still only able to target GCC due to assumptions about how compilers are used and behave. Distributed parallelism was also not fully possible at this stage beyond a small number of nodes, due to policies placing restrictions upon those using the university HPC depending on whether or not they had research funding allocated for CPU-cycles. Python was used to create the prototypes because it allowed development to take place rapidly, but was not expected to be suitable for the final implementation.

5.2.2 The Prototypes

Table 5.1 gives an overview of the timeline for these, with the major GCC releases and Intel microarchitectures (ignoring desktop and mobile chips). Chips should be assumed to be 64-bit x86 unless otherwise stated. It should be noted that although the prototypes span a time period from 2010 to 2016, in reality work on the final project, `OptSearch` began in February 2014. In table 5.1 the starting year is given as 2016, as this is when work dramatically picked up in pace and could progress without need for further input from prototyping.

Work was carried out in stages, with the identified requirements used to set specific milestones. At each milestone, more work was done with the prototypes to inform the next stages, so as to avoid the project becoming stale and falling so far behind the rapidly moving advances in compiler versions and CPU architectures as to be come useless.

The earliest work began as a hobby project in approximately August 2004, more than two years before the author officially registered as a PhD student, working on two generations of Intel Pentium 4 processors (Northwood and Prescott) and on AMD Athlon64. There were additionally several intervals when the work had to be halted, including for most of 2015 and part of 2016, during which time the PhD was suspended.

Prototype 1 (Python/Simple)**.** *In the first prototype, developed using Python 2.6 on* `Aquila` *at the University of Bath (see appendix A.2.1), the chosen optimization algorithm was a highly simplistic linear search [107]. While quick and easy to write, this turned out to be a poor choice even with a small subset of compiler flags. The search always failed due to the nature of the search space, and the frequency of local minima, which was far greater than expected at the start of the project.*

With this prototype, the search space could only be explored by manual intervention. Whenever a local minimum was encountered, this would be noted and the prototype adjusted to avoid it. This was considered acceptable for an early prototype, but clearly undesirable for a final implementation.

It was not only the speed of the search that was a problem. With prototype 1, several of the tuning runs were found to simply not terminate, or to fail with an error, due to the decisions made by the compiler during optimization. Similar failures were encountered during the compilation itself. This meant that few meaningful results could be obtained.

A similar search space is described in [6]. Although the authors use their own adaptive compiler, it uses what has become a familiar break down of optimization types in compiler design. This suggests that a search space of this nature should be expected with all modern compilers.

Prototype 2 (Python/Fixed-Random)**.** *This prototype was based heavily on the code in prototype 1. It prototype attempted to improve on the earlier prototype by replacing the search algorithm with a naively implemented fixed step size random search [149].*

It did not scale well to a large search space, requiring more memory than could be allowed with a benchmark running on the same machine, and was slow to converge.

Prototype 3 (Python/NAG)**.** *Prototype 2 was modified, replacing the optimization algorithm with a call to the NAG library [129], routine* `E05JBF` *([127]). This took some time to implement, due to the difficulties of interfacing between Python 2.6 and Fortran 77. This prototype performed better than the earlier ones, but struggled with the frequently encountered local minima and would often become stuck.*

It was not expected that the NAG library would be used in the final design, as it would be hampered by the limited portability of this proprietary library, but it again served as a useful example. Despite the problems of interfacing with a Fortran 77 library, it was faster to implement this interface than to write and test a similar search algorithm written from scratch. It was also reassuring to know that the one being used has been carefully tested.

A simple state machine giving some indication of the workings of prototype 3 can be seen in figure 5-1.

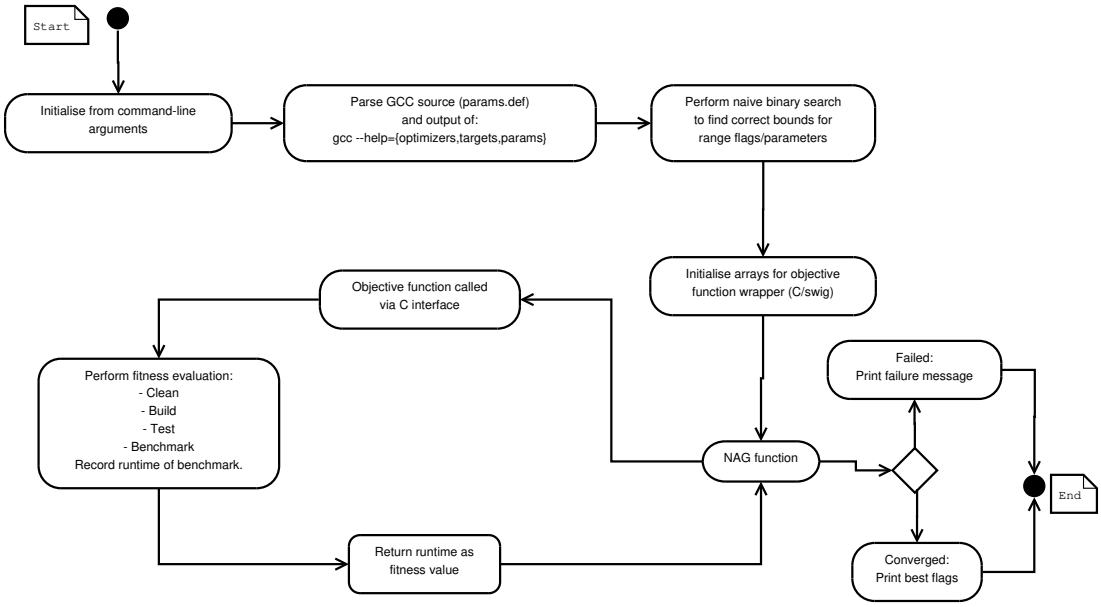

Figure 5-1: *State transitions of prototype 3*

Prototype 4 (Python27/Random-Random)**.** *Prototype 2 was rewritten in Python 2.7, with the search algorithm replaced with a recursive (random step size) random search, based on a simplification of [182]. This prototype performed better, avoiding the problems caused by frequently encountering local minima, but did not perform as well as hoped. Memory usage, and interference with the benchmark, were again a problem due to the difficulties in controlling the Python interpreter's use of memory, etc.*

These early prototypes were simplistic in nature, making no use of multithreading or MPI. They ran on a variety of systems, most of them servers of varying performance, since at the

time `Aquila` was the only cluster the author initially officially had access to[4].

These early prototypes were created with the sole intention of informing the author and helping to demonstrate that a similar technique could be viable. It was not expected that their exact approach would be appropriate in the final version. As well as the problems of interference from the search algorithm, the search itself was still very slow, with assessments carried out at many, many points in the search space (the benchmark being the most time-consuming part of the overall runtime), and could not guarantee finding an answer. Instead it served as a useful exercise, giving a better understanding of the search space. This was necessary to guide the choice of a suitable search algorithm later.

Several search algorithms were investigated and discarded during the development of these prototypes. Hill-climbing was one of these; it was ruled out once it became clear that the search space contains many local minima. Similarly, gradient descent and simulated annealing were also ruled out as approaches due to a tendency to take a very long time to leave a local minimum once encountered, if they leave it at all. A random search approach was initially very attractive as it is fast to implement and unlikely to be prone to such problems. Unlike most machine-learning approaches, it does not require a body of training data to be collected, a task that was likely to be time-consuming and which would have the problem of data storage becoming a problem[5].

Work at this time moved to `Iridis 3` and `Iridis 4` at the University of Southampton (see appendix A.2.2 and A.2.3).

Prototype 5 (Python27/OpenTuner)**.** *This prototype took a different approach, building on a simple example written in Python 2.7 by Ansel and now included with OpenTuner [8]. This was also not designed to be distributed across multiple compute nodes, and in fact the run time was very long as Ansel chose to use a genetic algorithm for searching.*

Later in the development of this prototype, Ansel also provided some other search options, including an implementation of Standard Particle Swarm Optimization (SPSO) [33], but this was not until after the decision had been made to abandon this approach. However, the genetic search did not suffer with the problems that the first two prototypes had suffered from, and produced reasonable results in line with those seen in [8].

One major difficulty with prototype 5 was that it made use of Python subprocesses to run in multiple threads. This meant that the obtained benchmark results were highly susceptible to interference from other threads running the compilation or benchmarking phases. This was expected at the time, but the use of subprocesses pervades the design of OpenTuner, and could not be easily avoided without significant work. In addition, this prototype took a very long time (2-3 weeks) to converge, too long (in the author's opinion) to justify its use on production machines, unless the performance gains are large.

A greater problem lay with the choice of Python for implementation, something that had already proven problematic in the earlier prototypes. At the time, of the machines the author had access to, prototype 5 could only be used on `Iridis 4`, as it required a later version of the operating system than was available elsewhere due to the large number of Python 2.7 libraries used by OpenTuner. Some of these would not build from source successfully on the older machines.

[4]The Maths dept had its own small HPCs, maintained by the author, but these were not used for her own research.

[5]At this time, the author was noticed to be the largest user of file system space on `Aquila`, where space was at a premium.

Prototype 6 (Python27/MPI)**.** *The final prototype was an evolution of the previous one, using MPI to distribute the workload, instead of Python subprocesses, and added checkpointing so that the previously 2-3 week runs could be shortened to a mere 4-6 days, and broken up without risking losing valuable CPU cycles to system failures or maintenance windows, etc. It also avoided interference with the benchmark by moving the fitness evaluation to a dedicated node, keeping the optimization algorithm separate.*

The demonstrated gains helped to emphasise the necessity of a more scalable approach, as the previous long run-time would have made tuning libraries for real world use impractical for most HPC centres.

Attempts were made to use both prototype 5 and prototype 6 on Southampton's `Amber` cluster (appendix A.2.4). These were unsuccessful, due a dearth of Python modules on that architecture, many of which could not be built from source without modification. This was confirmation, if any were needed, that Python was not going to be suitable for a portable auto-tuner of this nature.

Further project goals

With the understanding gleaned from these prototypes, it was felt that enough had been learnt about the problem space, and the right type of approach, to begin work on the tuner proper. In particular we had added three further high-level goals:

(iv) It shall be possible to distribute the search workload, so that the search can complete in a reasonable time

(v) There shall be no interference between evaluations of different points in the search space

(vi) The tool should be robust against compiler bugs

The interaction between these first two is part of the difficulty of the whole project.

Date	GCC Release	Chip release (best guess at availability)	Prototype begun
2002-01	-	Intel Northwood	First recorded manual experiments
2003-05	3.3.0		
2003-09	-	AMD Athlon64	
2004-02	-	Intel Prescott	
2004-08	-		First attempt at automation
2005-04	4.0.0		
2006-02	4.1.0		
2007-04	-	Intel Penryn	Beginning of PhD project
2007-05	4.2.0		
2007-09	-	AMD Barcelona	
2008-03	4.3.0		
2008-04	-	AMD Budapest	
2008-11	-	AMD Shanghai; Intel Nehalem	
2009-04	4.4.0		
2009-06	-	AMD Istanbul	
2010-01	-	Intel Westmere	
2010-03	-	AMD Magny-Cours; IBM POWER7 (PowerPC)	
2010-04	4.5.0		Prototype 1
2010-06	-	AMD Lisbon	
2011-01	-	AMD Valencia; Intel SandyBridge	
2011-02	4.6.0		Prototype 2
2011-10	-		Prototype 3
2011-11	-	AMD Interlagos	
2012-03	4.7.0	AMD Zurich	
2012-04	-	Intel IvyBridge	Prototype 4
2012-11	-	AMD Abu Dhabi, Delhi	
2012-12	-	AMD Warsaw, Seoul	
2013-03	4.8.0	APM X-Gene 1 (ARMv8)	
2013-06	-	Intel Haswell	
2014-04	4.9.0		Prototype 5
2014-06	-	IBM POWER8 (PowerPC)	
2015-02	-	Intel Broadwell	
2015-04	5.1.0		
2015-07	5.2.0		
2015-08	-	Intel Skylake	
2015-12	5.3.0		
2016-03	5.4.0		Prototype 6
2016-04	6.1.0		
2016-08	6.2.0	IBM POWER9 (PowerPC)	
2016-10	-		Final version (`OptSearch`)
2016-12	6.3.0		
2017-02	-	Intel Kaby Lake	
2017-03	-	Cavium ThunderX2 (ARMv8)	
2017-04	6.4.0		
2017-05	7.1.0		
2017-06	-	AMD Epyc, Ryzen	
2017-08	7.2.0		
2017-10	5.5.0		
2018-01	7.3.0		
2018-05	8.1.0		
2018-07	8.2.0		

Table 5.1: *Timeline of major GCC releases, approximate dates of chip availability, and beginning of work on prototypes.*

CHAPTER 6

THE CONVERGENCE OF BENCHMARKS

In parallel with the prototyping work, it was necessary to both determine the baseline per-node performance of the target systems (an activity that was repeated several times with each machine as the project evolved), as well as the behaviour of the benchmark with the target library on those machines. This information was used to guide the prototyping process and to make several of the decisions described in chapter 7.

6.1 Introduction

The practice of benchmarking is well established in supercomputing circles. It is human nature to want to know whether one machine outperforms another, and if nothing else the TOP500 list is evidence of this. Analysis of benchmark results can also indicate the presence of subtle (and less subtle) hardware and software problems that may not otherwise be noticed before a supercomputer is made available to the user community. Similar analysis is vital after software or hardware changes are made, to monitor any potential changes in performance (for example, those introduced by security patches, as was seen with the fixes for the Spectre and Meltdown [118] vulnerabilities).

Generally, benchmarks are run an arbitrary number of times, and a result may be picked from this set, its choice varying depending on the purpose of the exercise. When evaluating the results, a judgement must be made by the benchmarker as to whether the results are within expected bounds, and whether they are close enough to one another to be relied upon. This is especially important on modern hardware, as results vary increasingly from one run to another [29], provoking calls for ways to ensure consistency [109].

For this project, such a process must be automated. It is simply not practical for a human to be involved in the benchmarking step of an auto-tuner; it must happen too frequently, with each result used to determine the next point in the search space to evaluate. It is also not clear how often human error means that a performance benchmark has been considered to have converged incorrectly. This is something that is rarely mentioned in benchmarking circles. It is discussed briefly in [32], in relation to judging performance of optimization algorithms, but otherwise appears to be an area that has been largely ignored.

In this chapter, understanding of the process is sought, and the typical results explored with a view to automatically determining whether a benchmark can be considered to have converged on a particular result. This exercise also fulfils a prerequisite of any auto-tuning work: That of

providing a figure for the baseline performance of the target software on the target hardware. Known in the industry as baselining, this is a required first step before any performance tuning or optimization work can begin.

6.1.1 Background

It is common practice to run a benchmark several times, look at the results to ascertain their closeness to one another, pick one (good practice would choose the median, but it is not unknown for someone to quote the best of the set) and use that as the result.

This is not to say that the practice is invalid, though it is difficult to be certain that a benchmark has truly converged. This is becoming increasingly difficult on modern hardware, where the method of categorising chips known as binning tends to take into account power consumption, rather than observed clock speed. This can result in chips that are supposedly identical differing in clock speed by as much as 100MHz, in addition to selecting differing per-core "Turbo" frequencies, as given in [91] and other specification updates from this and other manufacturers.

Furthermore, as it is no longer practical to increase the clock speed of a chip and power consumption must be constrained, optimizations and methods of keeping power consumption low mean that we increasingly see variation even in timing [148], whether measured in cycle count or μs, between the same small set of instructions being computed. This may be due to some parts of the chip, such as the upper 128-bit half of the vector instruction execution units on Intel's Skylake processors, being powered off at the slightest suggestion of not being required, as described in section 2.3. In this and other chips, there are also differences in how μops are fused (or not), which results in different numbers of μops per cycle for what ought to be the same task. For this, the blame might be laid at the feet of such optimizations as out of order and speculative execution. Once the province of compilers, such optimizations are now normal in hardware.

These and similar features are likely to remain with us, and to increase in number, as time marches on. It is necessary for any benchmarking process to take into account this variation, which will affect different applications in ways that are different and difficult, perhaps impossible, to predict.

To know that a benchmark has converged, one must first know how much each result from each run of the benchmark is expected to differ from others, both on the same hardware and on other, supposedly identical hardware.

If we are to automate the process, it is necessary to ensure that convergence can be measured and checked. A statistical approach would also avoid the problem of human error in determining whether a benchmark truly has converged, and whether the variation in results is within acceptable/expected bounds.

How many times should a benchmark be run? Ideally, one would define an acceptable variation, ε, and then proceed in running the benchmark as many times as necessary to achieve an observed variation that is below this threshold. In the real world, we cannot simply run the benchmark forever, so a maximum number of runs must be defined. Any set of n runs for which the variation is above the threshold must be discarded, as their results are meaningless.

Of course, it is difficult to know what n will be for any given problem on any given machine, and it will very probably be different for every combination of problem (benchmark, libraries, compiler) and machine. Therefore it is necessary to determine what ε and n are for a particular benchmark before tuning it. The only way to do this is to choose a very large n, as large as is practical, run the benchmark n times and examine the results.

In the course of benchmarking a system, the early results are often discarded. This is because they are usually slightly lower than those seen later, due to the wake-up times required in modern processors. As explained in section 2.3, many functional units of modern CPUs power down if not in constant demand. Arguments for and against including these early results centre on the purpose of the benchmark and how its results will be used. Seeking a more realistic reflection of the user experience might suggest that it is better to include them, but if the goal is a position in the TOP500, the impact of their inclusion could cost the machine a place, causing it to be ranked lower than a competitor.

6.1.2 Method

This was a fairly simple experiment. The reference BLAS [117] was compiled with GCC [154] with `-O0`, `-O1`, `-O2` and `-O3`. Each resulting library was then used for 20 runs of the high performance LINPACK benchmark (HPL) [136] on a single node.

20 was chosen because it seems intuitively to be high enough, and it is the number frequently used by others when benchmarking. As in traditional benchmarking, the results were also looked at to check that, to the eye, they appear to be "close enough".

Machine noise was minimised as much as possible by ensuring sole use of the node on which the benchmark was running. By spreading the workload across several machines, it was also possible to check that no one machine was noisier than the others (within sensible limits). With single machines used for all runs, network noise, especially that caused by the jobs of others, could be more or less avoided entirely.

To ensure that any compiler optimization bugs were noticed, the experiment was repeated both with and without recompiling the BLAS library each time the benchmark was run.

The results were then analysed using MATLAB's statistical routines. This gave an indication of what to expect, particularly in terms of standard deviation, when benchmarking within the auto-tuner.

6.1.3 Environmental constraints

Due to the restrictions on the "free" account used on `Balena`, parameters for HPL were chosen such that the longest run, that resulting from compiling with `-O0`, would finish with the maximum 6 hour walltime. This allowed for a problem size that would exercise main memory, rather than being contained within the processor cache. To ensure such a large problem size, the number of algorithms usually chosen in the default input file to HPL was reduced to just one, chosen arbitrarily. Preliminary experiments had already demonstrated that there is little difference between them, and certainly not enough to suggest one over the other for this type of experiment.

HPL was configured to use right-looking LU factorisation for both the main and recursive panel factorisation. The contents of the input file used is available in A copy of the HPL.dat used is available in appendix C.

6.1.4 Results

Initial results from `Balena` were unexpected. Intel®IvyBridge processors are expected to vary a little between one another, even within the same stepping, as are all modern processors. However, this could not account for the massive variation in performance between the results

from each node. At least two populations of nodes could immediately be discerned from the results, and closer inspection revealed a third, clearly visible in figure 6-1.

The differences are more obvious if each dataset is plotted separately, as in figure 6-2. One node appeared to be faulty, as it was performing very differently to its peers.

On discussion with the system administrators responsible for the machine, it became clear that the differences seen in figure 6-1 were due to hardware differences, rather than errors. The compute nodes could be grouped by the type of RAM installed, their differences clearly visible in the HPL results:

1. `node-sw-[001-080]` have 16 8GB single ranked DIMMS at 1866MHz
2. `node-sw-[081-164]` have 8 8GB dual ranked DIMMS at 1866MHz
3. `node-sw-fat-[001-002]` have 32 32GB quad ranked DIMMS at 1333MHz (not included in the results)

The nodes could be further broken down within those groupings, again by performance, with the difference slightly less marked (though visible in the graph). These differences were due to the DIMMs having been supplied by 3 manufacturers:

1. Hynix:
   `node-dw-ngpu-[001-003,005]`,
   `node-dw-phi-006`,
   `node-sky-[001-014,016]`,
   `node-sw-[086,097-120,125]`
2. Samsung:
   `node-as-ngpu-[001-007]`,
   `node-as-phi-[001-002]`,
   `node-dw-phi-[001-004]`,
   `node-nvme-[001-002]`,
   `node-sw-[001-022,081-085,087-096,121-129,131-168]`,
   `node-sw-fat-[001-002]`
3. Micron:
   `node-dw-ngpu-004`,
   `node-dw-phi-[005,007]`,
   `node-sw-[023-080,091]`

Interestingly, although nodes with Hynix DIMMs performed slightly better than their peers, the variation within their results was higher. In both cases the difference is so small as to be unimportant in the vast majority of use cases.

In addition, one node, a clear outlier initially thought to be faulty, was discovered to have been installed with a mixture of dual and single ranked DIMMs. This was drawn to the attention of the suppliers, and the problem quickly rectified. Astonishingly, it was explained that no one had noticed in the previous 3 years of the system being in service, and so the system administrators saw no benefit in making the information about differences between the installed DIMMs available to the general user population.

With this new information, selecting a subset of nodes with identical DIMMs gives a population with a performance that is closer to normal, as shown in figure 6-4 and figure 6-5. This provides a better set of candidates for future experiments.

Method	Mean runtime (s)	Std dev	Std Dev as % of mean
`-O0`	95.40s	0.06s	0.06
`-O1`	26.74s	0.02s	0.07
`-O2`	17.10s	0.02s	0.12
`-O3`	13.08s	0.03s	0.23
`-Ofast`	13.06s	0.04s	0.31
ATLAS	2.99s	0.01s	0.33

Table 6.1: *Standard deviation of HPL at different optimization levels (subset of IvyBridge nodes)*

Although both mean performance and standard deviation differed mildly between manufacturers, a far greater difference is observed (and expected) between differently ranked DIMMs of different speeds. The author is surprised that this is not of interest to the general users, since any whose distributed jobs run across a mixture of nodes will be affected by the differing speeds. It is difficult to say why they have not noticed, but it may be due to an inability to determine what performance should be expected from a previously unencountered system. Nevertheless, if the mixture of nodes varies, the execution speed would likely also vary, and a casual inspection of results (assuming that they are repeated) should make this clear. It is strange that no one would have noticed this, though in the author's experience, many who find problems never take the time to notify the system administrators, unless their work cannot continue without those problems being addressed.

While the Skylake nodes are thought to be homogeneous, it is notable that figure 6-6 does not show a normal distribution. Rather, the distribution appears skewed, with a long left-hand tail.

As can be seen from table 6.1, which lists values for standard deviation for a subset of nodes on `Balena` (58 nodes with the same memory type: `node-sw-[023-080]`), standard deviation varies slightly at different optimization levels. In these results, it can be seen to increase as a percentage of the mean runtime (in seconds) as the level of optimization increases. Process pinning was used for all of these results, which slightly reduced the variation and improved performance.

These results were obtained from single machines (but not a single machine, though there appears to be no correlation). No compiler optimization bugs were observed, as the results differed very little between runs where the BLAS library was recompiled prior to running the benchmark every time and those runs without recompilation.

This table also shows how stable a benchmark HPL is. Other benchmarks (and user code) has been observed to suffer from greater degrees of variation, whatever amount of optimization is employed during compilation.This suggests its use (or that of `DGEMM`) in diagnosing subtle hardware problems in CPUs, as is alluded to in [150, page 4]. The author has previously used the HPL benchmark (among others) both during acceptance testing and for fault finding.

Although it appears from this that the performance of the reference BLAS is not great, these results alone are not a true reflection of the performance of this BLAS (or of `Balena`). For this reason table 6.1 includes a line for ATLAS [178], which uses a different implementation of `DGEMM` (chosen at build time from a suite of parameterised kernels) and is known to perform well.

Using ATLAS for comparison, it can be seen that even with `-Ofast` and a mature compiler (GCC 7.3.0), the reference BLAS performs poorly, with HPL taking more than 4 times as long to run to completion.

A single Ivybridge node of `Balena` has a R_{peak} of around 333 GFLOPS[1] (ignoring Turbo[2]). However, due the small problem used for these experiments, even ATLAS, which on `Balena` is able to achieve a runtime (in seconds) of over 90% of R_{peak}, cannot achieve this. Instead a large enough problem (N in the HPL.dat file used as input to HPL) would be required to approach R_{peak}.

The HPL.dat used for these experiments is available in appendix C. The small value for N is the largest that was usable (would finish within the allowed walltime even with the reference BLAS built with `-O0`) for a single node job in the non-paying queue on `Balena`. Larger problems could only be considered with higher optimization levels, due to the significant differences in runtime. Note however that with the largest possible N (sized to fill the available memory on the node in use), the runtime can be significantly longer than is practical for anything other than acceptance testing or similar activities, being measured in hours and days.

Note that differences in memory do not account for the noticeable performance fluctuations between the Skylake nodes, visible in figure 6-3. These differences remain regardless of how well optimized the library is by the compiler. In this case, not all cores are being used, and the benchmark is pinned to the first 16 cores. The DIMMs are from the same manufacturer in all but one node, and are of the same type, size and speed, so the variation is not due to the same issue that is seen on the IvyBridge nodes.

[1] 2.6GHz $\times$ 2FMA $\times$ 4SIMD AVX $\times$ 8cores $\times$ 2sockets

[2] Because of features like Turbo, which boost the clock frequency from 2.6GHz to 3.4GHz temporarily, R_{peak} is difficult to calculate accurately

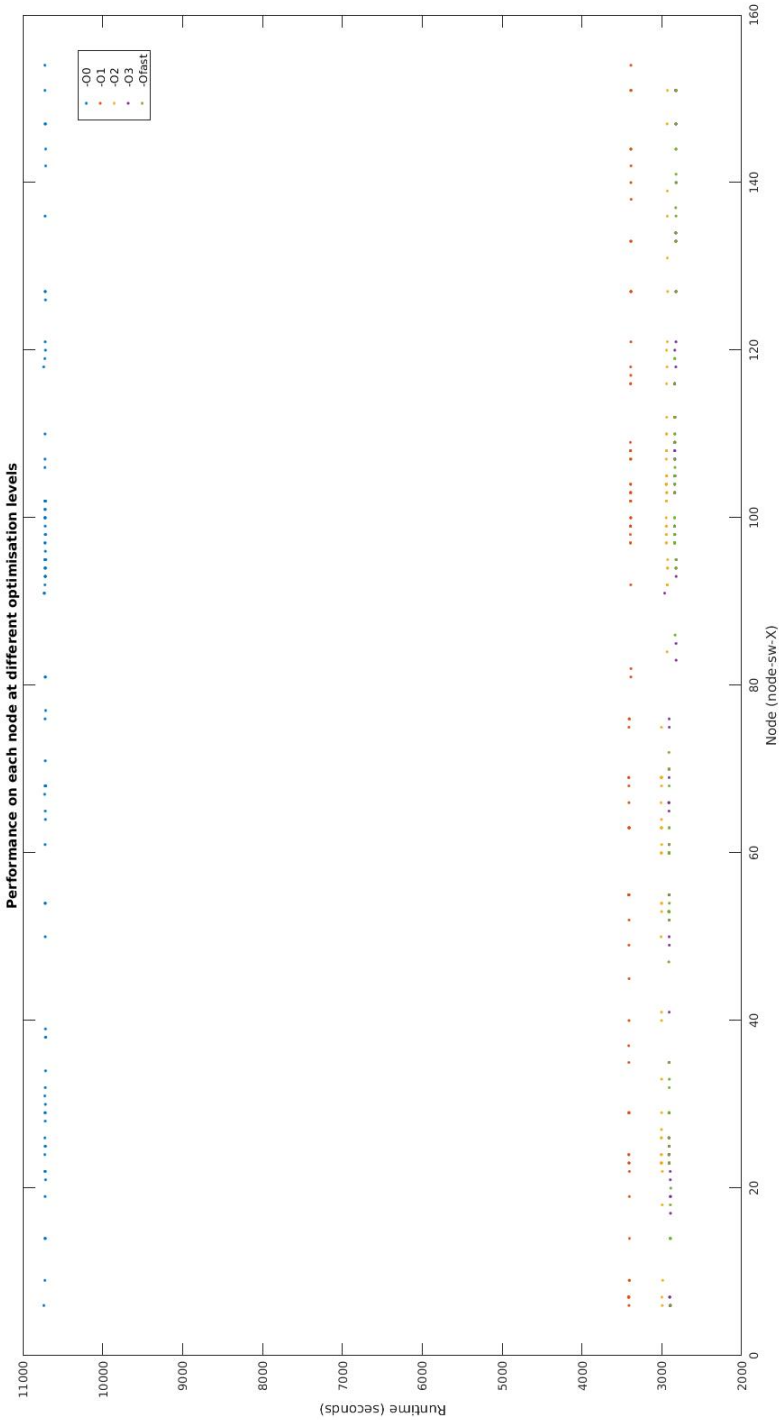

Figure 6-1: *Results from* `Balena`*, showing differences in performance across the IvyBridge nodes*

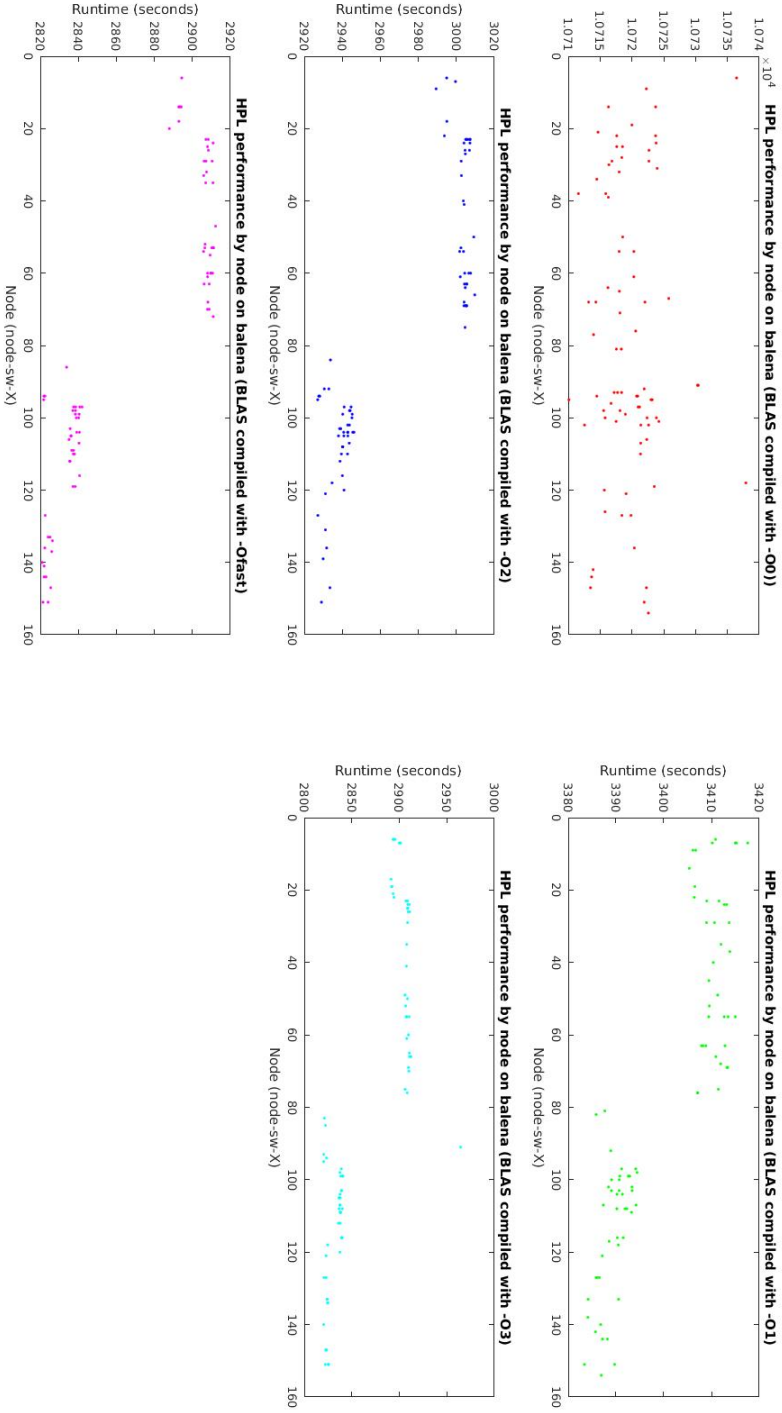

Figure 6-2: *Results from* `Balena`*, showing clear differences between the populations of IvyBridge nodes regardless of optimization level (same data as in figure 6-1)*

Figure 6-3: *Results from* `Balena`*, showing differences in performance across the skylake nodes at different optimization levels*

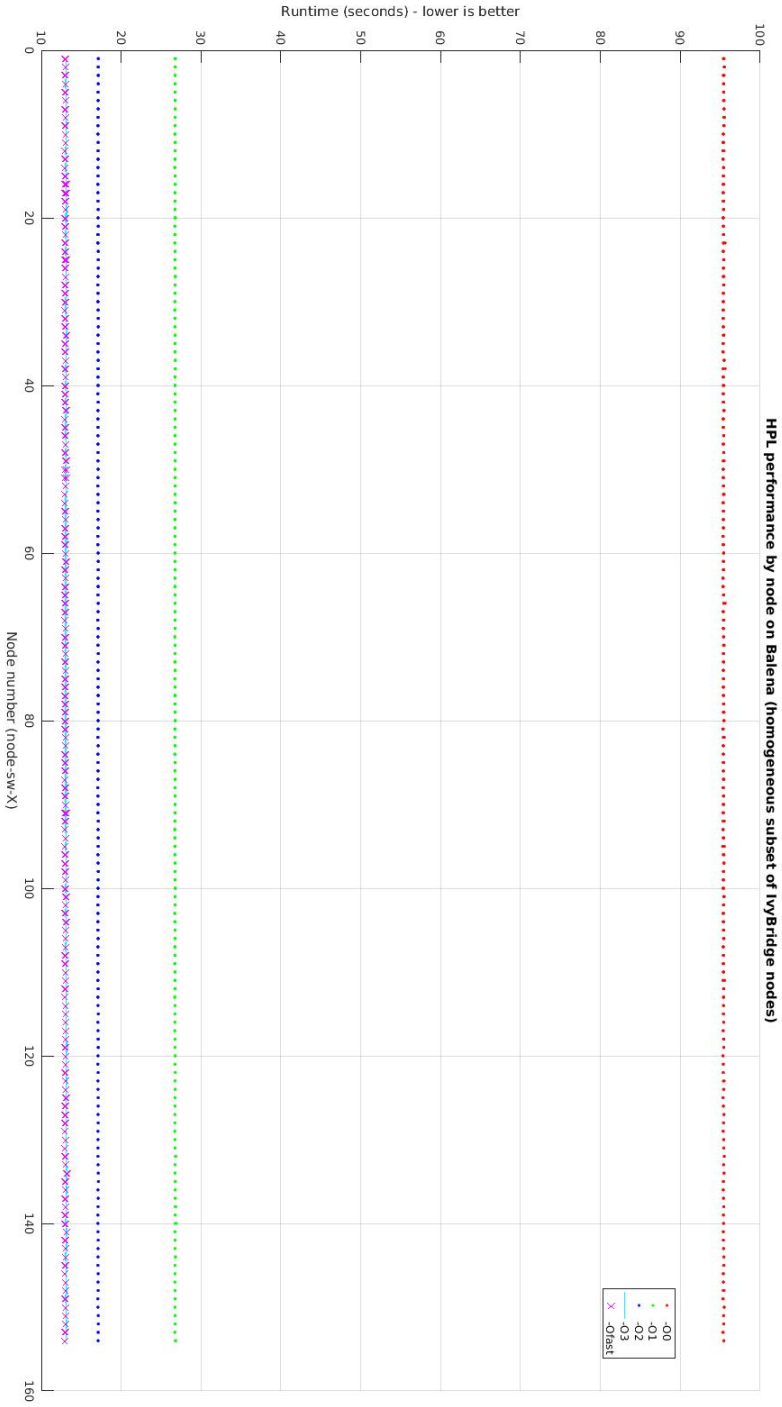

Figure 6-4: *Per-node HPL performance for a homogeneous subset of* `Balena` *IvyBridge nodes (Reference BLAS compiled with* `-O0`*)*

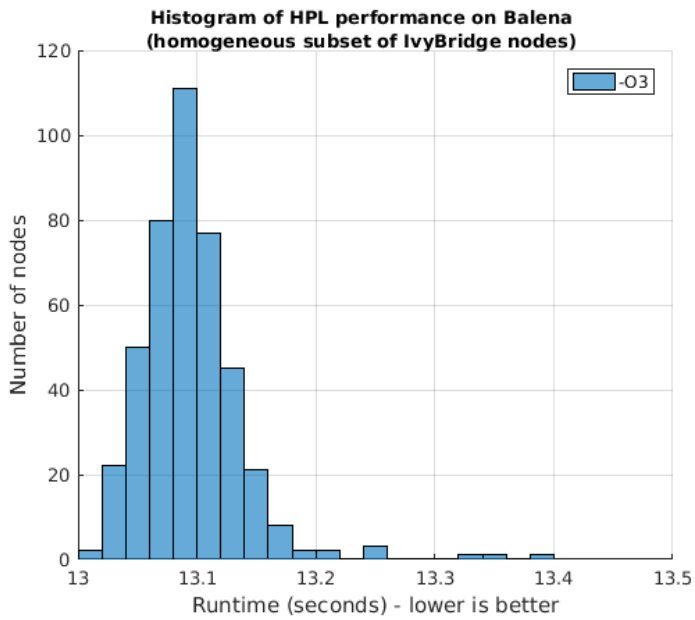

Figure 6-5: *Histogram of per-node HPL performance for a homogeneous subset of* `Balena` *IvyBridge nodes (Reference BLAS compiled with* `-O3`*)*

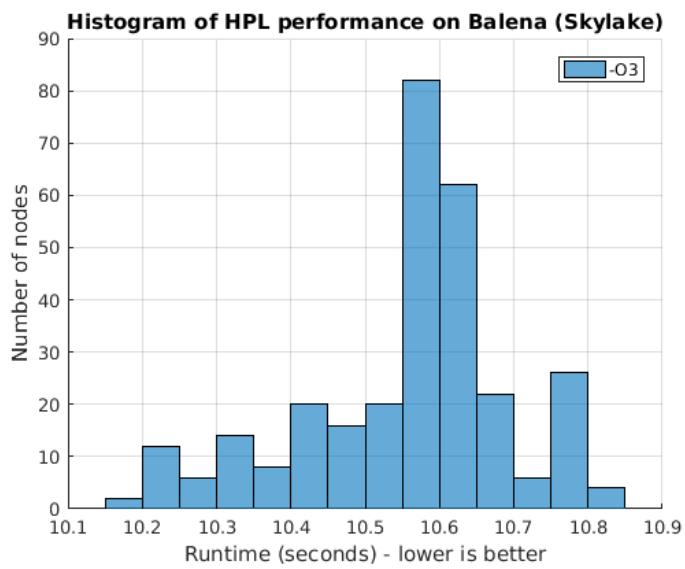

Figure 6-6: *Histogram of per-node HPL performance for* `Balena` *Skylake nodes (Reference BLAS compiled with* `-O3`*)*

6.1.5 Homogeneous machines are not always uniform

Contrary to what might be expected, homogeneous clusters do not guarantee "uniform" (realistically, within 1-2%) performance across all nodes within the cluster, nor should performance be assumed to be within specification just because the system is new and there are no obvious errors.

When a system is commissioned, the suppliers and/or the system administrators may spend an appreciable amount of time running tests and studying the results for anomalies that might give away an under-performing node[3], as we saw above. Unfortunately, this practice seems to be uncommon among the many suppliers of commodity clusters, and few system administrators are familiar with the practice of benchmarking or its practical applications for fault finding or early diagnosis of problems. Since their job is to keep everything running smoothly, a knowledge of this process could benefit both them and their user community, provided they had time to apply it.

In an ideal world, the benchmarking process would be followed during every maintenance cycle, ensuring that all nodes are performing within specification after any changes are made. However, this is rarely the case in practice, and not long after entering production it is common to see performance vary across the nodes by more than the specification of the system would allow. This often increases over time, and changes in nature as parts within the system are replaced.

Since this variation is relatively small, for the majority of users of the system this may be unimportant. This is especially true for those whose workloads entail large numbers of single node jobs that are not dependent on uniform hardware, as is the case in genomics, or where timing differences are unimportant.

Unfortunately, even slightly increased variation, and especially entirely different populations of nodes with very different behaviour, is very significant when benchmarking, and when assessing performance of subtle changes in compilation (or algorithm), as when auto-tuning. It is also a problem for most parallel codes, which are often slowed to the pace of the slowest node in the cohort assigned to them by the workload manager.

It should be noted that in figure 6-3, the results within individual nodes do not fluctuate any more than is expected for a single node. It should also be noted that, despite the variation in performance of individual nodes, they are operating with specification according to the chip manufacturer, Intel, as the variation between nodes is not greater than 10%. As an example, several figures in the specification update for this processor, particularly [92, Figure 7], list significant amounts of variation in clock frequency per core when Turbo mode is engaged. This varies depending on how many cores are in use.

Even the stable HPL benchmark exhibits greater variation when run across multiple nodes for this reason, and such variation may catch users of these systems off-guard if they are not aware of it.

6.1.6 Conclusion

Even when compiled with `-O3`, performance of the reference BLAS is low. This is not a revelation, as it has been known and stated many times by others.

While the performance of HPL increases with problem size, the poor performance cannot be blamed on the use of a small problem in this case. Rather, it can be seen that the reference

[3] Or over-performing, though this happens rarely and is more easily dealt with

BLAS, while easy to understand and maintain, does not deliver “good” performance. This is not its intended purpose.

Of more interest is the observation that standard deviation increases with the level of compiler optimization. This was expected, but has not to the author’s knowledge been previously documented.

It is also worth noting that preliminary experiments also showed that it is not necessary to use a large problem size for HPL to show differences between nodes, which is useful for those using benchmarking for diagnosis of problems within a short maintenance window. This means that, so long as the entire node memory need not be exercised, problems relating to the CPU or speed of memory access can be highlighted by only 5 data points from runs of HPL that each take around 60 seconds to complete.

A further expansion of this work would be to perform the same evaluation on other compilers, other machines and perhaps also other operating systems. A detailed comparison of how performance variation is affected by problem size (and node count) may be interesting to those benchmarking their systems. Other benchmarks, such as HPCG [45], could also make interesting candidates for examination.

An understanding of the amount of deviation to be expected when benchmarking is a requirement when automating the process. With this understanding, it should now be possible to judge programmatically whether or not a benchmark needs to be repeated many times. For an auto-tuner running on a supercomputer, where CPU hours are a precious commodity, this knowledge is not without value.

CHAPTER 7

DESIGN SPECIFICATION

> The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.
>
> Donald Knuth

7.1 Lessons drawn from prototyping

This project relies heavily on the successes and failures of its early prototypes. A summary of each of the key areas informed by this process is given below. These lessons were used to guide the final design.

7.1.1 Language choice

At this point, it was decided that Python and in particular the OpenTuner framework, while very helpful for prototyping, could not deliver the results that would ultimately be required. OpenTuner is extremely complicated, too complicated for our needs. It is difficult to debug and to extend, especially if one wishes to keep up with the changes made by Ansel, who is still actively working on the project. In addition, Python is not as efficient a language as the more traditional C [5], despite its popularity.

Although the OpenTuner author, Ansel, has put considerable work into the framework, it is not ideally suited to tuning this type of problem, nor is it particularly portable to emerging architectures. This was especially obvious when attempting to use the prototype on Southampton's `Amber` (ARM) cluster (see appendix A.2.4 on page 122) in 2014. Unfortunately, very few Python libraries were available, and building from source proved unworkable, not just due to the sheer number of modules required. Many relied on bindings for libraries that were not available for that architecture, and these could not be ported without considerable work.

In addition to the difficulties caused by the portability of Python and its many modules, there are several implicit assumptions clear within the design of the framework. It was not until over a year later, after exchanging emails with Ansel, that it looked like it might be possible to make modifications to ensure the tuner will run in a distributed fashion. He has also since made it much easier to add other search algorithms, so that the user is not obliged to use

only what is provided. These pre-written search algorithms are generally fairly simplistic and assume a continuous $\mathbb{R}$ space. It seems that, in the pursuit of satisfying all possible users, the framework has become extremely large and is difficult for the uninitiated to modify or extend, especially if their use-case is not quite what Ansel had in mind.

After several prototypes implemented in Python, it was decided that the final project would be implemented in C. This is a language familiar to the author and is already well used in the HPC world. In addition, it is the language used to implement the Linux operating system, so will be well supported on any emerging architecture. If the operating system is written in C, there must exist a C compiler that supports that architecture, something that cannot be guaranteed with other languages.

Linux is the operating system of choice in HPC at the current time, as shown by its continued dominance of the TOP500 list [122], and it seems this is likely to continue for many years. Any system with operating system support will have to have an existing C compiler and standard libraries, so by writing an application that is self-contained (few, if any, dependencies on external libraries) and adhering to accepted standards (such as POSIX) it should be possible to ensure portability to the largest number of architectures.

Prototyping demonstrated that despite being much faster to write, Python is both slow to run and introduces library dependencies that can be difficult to manage. Less commonly used architectures tend to be left without key modules, and for a project aimed at emerging platforms, this is a serious failing.

Even though the slowness of the Python interpreter can be worked around, for example, by compiling all the Python with Cython, the resulting binary is still larger than necessary and has been observed to be generally noisier, introducing jitter in the form of non-ideal memory accesses (leading to an increase in cache misses), than a comparable program written directly in C by an experienced C programmer. This noise is highly undesirable in any benchmarking exercise due to the interference it causes, and the difficulty in attributing performance issues to it.

Compiling the software to an executable via Cython and GCC is extremely difficult due to the large numbers of dependencies. In addition, not all Python modules used by OpenTuner will run on all the platforms targeted in this project. POWER8 and ARMv8 were experimented on, but at that time the Python dependencies could not be satisfied and so ultimately OpenTuner was unusable. This is extremely undesirable in a tool intended to target emerging architectures.

While C++ is popular, it is not as well supported as C on very new architectures. In addition, it does not confer many advantages beyond the language features that arguably may make it easier to use (or more difficult to maintain, or compile efficiently). The system libraries on which this software will rely are all written in C, and few MPI implementations include C++ interfaces.

Fortran is still a very popular choice for numerical codes in HPC, but I/O is more difficult than in C, calling system libraries (written in C) is more difficult. As with C++, it is unlikely to be as well supported on the most bleeding edge architectures, which is where this tool is intended to be used.

In recent years, functional programming languages have seen a resurgence. The approach these languages facilitate is extremely attractive to the purist, but the experience of the author is that all the currently popular functional language implementations (GHC [159], Racket [163], Chicken [158], for example) are extremely difficult to bootstrap without a functioning version of that compiler. They are, as a result, rarely available for emerging architectures. They also share with Python the difficulty of obtaining usable modules in a timely fashion on such architectures.

Other languages suffer from the these and similar problems, with interpreted languages being the least desirable due to the additional noise generated by the interpreter. They are also often difficult to manage where use of system resources, such as memory, are concerned.

The chosen implementation language should ideally make it easy to achieve a small memory footprint, something that can be very challenging in Python, and should be well supported on architectures on UNIX and Linux operating systems where not all libraries may be available yet. This is considerably easier to achieve in C than in most other languages.

UNIXes also make heavy use of C, and with standards such as POSIX it should not be too difficult to ensure that a tool written in C will work on those too, should the landscape change.

7.1.2 Choice of optimization algorithm

These problems meant that a robust optimization algorithm had to be chosen. Initially one of the NAG optimization routines had appeared attractive, especially considering that, being a commercially available library, it should have been well tested. However, it was not robust enough to cope with this type of frequent failure. Several other optimisation algorithms were considered, but on close inspection were deemed unlikely to be suitable for this search space.

Some machine learning algorithms (particularly some of the simpler supervised learning approaches [27], due to the shorter implementation time) were considered, but due to the training requirements these were also considered to be unrealistic to implement. Significant time would be required to generate and use enough training data to be useful, something that would be required for every target architecture and every target compiler on that architecture. The barrier to entry for those expected to use the tool would be much higher unless this time-consuming process could be automated. Creating the means to automate this data generation (and dimension reduction, where possible) would likely add considerably to the implementation time of the project.

The OpenTuner [8] author, Ansel, had suggested using a genetic algorithm, a popular choice for compiler optimisation tuning, and this is what he used in his simplified example. These tend to be slow-running and somewhat clumsy as an approach, requiring many mutations (and hence many assessments of points in the search space) before a successful result is found. With the benchmark being the most time-consuming part of each fitness evaluation, reducing the number of times this might need to be run before convergence on a suitable result is an attractive way to ensure a search completes in a reasonable amount of time.

This led the author to search for something with the robustness of a genetic algorithm, but which potentially requires a smaller number of fitness evaluations. One attractive option was a particle swarm optimizer. After some success in prototyping (targeting small, artificial problems due to time constraints), this was chosen for the final implementation, with the intention of possibly switching later to an asynchronous parallel variant, similar to that described in [151], should it prove necessary.

PSO has already been compared favourably with other search algorithms considered during prototyping [152]. In this comparison, the authors helpfully also provide another argument in favour of a search algorithm of this type:

> "the selection of compiler flags also affects the selection of the best code generation parameters, so we found that it is useful to tune them in conjunction with each other."

In using PSO, each compiler flag is mapped to a dimension in which to perform the search. Particles are moved around within the resulting search space, allowing the adjustment of several

dimensions (and so flag/parameter values) with each search iteration. Implementations differ in how particles reaching, or overshooting, the boundary are treated. In [30] it was shown that the method of confining particles can have a significant impact on how successful the search is.

Initially it was thought that an off-the-shelf implementation might be suitable, but on investigation it was found that none are quite what is needed. Even those targeting search spaces with many dimensions, such as [77], did not consider as many dimensions as are required for this thesis. (In [77], 40 is presented as an example of a high number.)

This led to the author implementing a slightly modified version of [33], similar to that described in [135]. This is used to feed the work queue of a task farm. Using MPI, this task farm then distributes work across a number of nodes. The returned results of fitness evaluations are used to drive this implementation of SPSO, rather than a loop, making it asynchronous. With the differing runtimes of the benchmark with different optimisations applied, it should also guarantee a level of randomness as recommended in [135]. Ideal parameters have to be discovered by experimentation, but those recommended in [33] were used initially.

The established methods also almost exclusively target problems in *Real* search spaces, unlike what is required here. This, and the non-continuous dimensions created by using them as a proxy for compiler flags, has further implications for how particles that reach the search space bounds are treated. With many very short dimensions, where the corresponding coordinate can have a value only within the interval $[-1, 1]$ (with -1 generally corresponding to the compiler default[1], while the other two values represent the two states of a boolean), particles are prone to overshooting the boundaries at every iteration. The choice in such cases is simply either "change" or "no change" (most often 0 or 1). Never changing means never getting anywhere. Although it might initially appear attractive to concatenate several such dimensions into one with multiple values, this does not work when calculating the distances or velocities/displacements required by each iteration of the search.

Boundary conditions

At boundaries, several strategies have been suggested. Initially, it was considered preferable to bounce back in, rather than use those (such as shr or shrink) suggested in [77] and illustrated in fig 1 of that paper).

With a discrete search space in $\mathbb{Z}$ rather than a continuous search space in $\mathbb{R}$, the effects of conversion from values in $\mathbb{R}$ to values in $\mathbb{Z}$ that are within the very short bounds of the search dimensions results in particles meeting the boundaries rather more frequently than they might in other search problems (in continuous $\mathbb{R}$ space) might. The additional difficulty presented by many of the dimensional spaces being very small, with a valid range being perhaps between $[0, 3)$, results in leaving the search space at each update being increasingly likely.

Thus a more robust method of dealing with the boundaries is necessary. Simply not moving for an iteration, one method suggested by [77], would leave a large number of no-op cycles, wasting a considerable amount of precious time[2]. There is also no guarantee that a particle on the boundary would return to the search space in future iterations.

The initially chosen approach instead treats the particle as a ball (or similar to rays in ray tracing), with the boundary considered as though a hard object or wall. This was the most

[1] A poorly documented feature of the default values is that they may set or unset other values; by not setting a value at all, the compiler is allowed to make its own choice.

[2] Time is most definitely money in the world of HPC

intuitive way to address the problem, but it met with several difficulties in implementation, and without dampening the particle would bounce around indefinitely.

As this approach did not alleviate the problem of becoming stuck within a subspace, it was decided to use another approach. This one, less well known in SPSO implementations, is known as periodic boundary conditions [52], and is a well established, commonly used technique for molecular dynamics simulation. It was also hoped that, by using this method, which is used in molecular dynamics to "fool" particles in a small simulation into behaving as though they are in a larger simulated area, would help to offset the difficulties caused by the short dimensions of this discrete search space. In this case, the simplest approach of restricting particles to the bounding box (the search space) was chosen.

7.1.3 Repetition of benchmark

Very little can be found in the literature regarding how many times a benchmark problem can be run before convergence on a result. Clerc [32] seems to make the only real comment on repetition of runs in this context, and does not suggest a solution.

General practice would seem to be to pick a number arbitrarily ($[3,20]$ being a popular range). Those experienced in benchmarking have likely observed that established benchmarks have, in the past, exhibited what might be termed "uniform" performance, with low standard deviation. Many benchmarks have become established precisely because of this behaviour; a benchmark with unpredictable performance is less useful.

Unfortunately, performance variation can (and does) come from a variety of sources.

7.1.4 Choice of PRNG

The choice of optimizer is heavily dependent on the selection of a suitable PRNG [32]. Further, results should be repeatable if possible (though an asynchronous approach may make this impossible to guarantee), so cryptographic randomness is neither required nor desirable.

Changing the PRNG in the future is likely to be necessary if a better design is found, so the code should allow for this.

One potential area of concern is that the vast majority of PRNG implementations produce a `double`, generally in $[0,1)$. Adjusting to produce an `int` (or `long`) that is within a very small interval, e.g. $[-1,1]$ is difficult to do without introducing bias.

7.1.5 Fault Tolerance

When individual nodes fail it can be difficult for any running application to detect, and doing so is considered to be beyond the scope of this project. Usually, a single MPI rank failing (e.g. by calling `MPI_Abort()` or by the MPI implementation detecting a non-zero exit code) will result in the termination of the job. This is the simplistic way that failures are handled by the majority of supercomputing applications, with check-pointing or some other method of resuming from a previous point the method of failure recovery.

Hard errors of this type are not a worry, but soft errors may occur and go undetected during the search. These could affect the results, if some of those search locations are affected by being evaluated on a faulty compute node. It is considered to be almost impossible to determine this from within the tuner itself, and so for a search to be successful it is often necessary to rely on the systems administrator to check and deal with any soft errors. Since the expected user of

such a tuner is the systems administrator or application specialist, it is hoped that this is not an unreasonable expectation.

Unfortunately, the study in chapter 6 would suggest that on the majority of systems it may be too much to hope for. As a mitigation, it is recommended that all nodes are baselined, similar to what was done in chapter 6, before beginning any tuning exercise.

7.2 Key design decisions

In light of the considerable investigation undertaken during analysis of the requirements, the several decisions were made about the necessary behaviour of the software, and its form, if the requirements are to be satisfied.

Figure 7-1 shows a very high level view of how the software works, with compilation and fitness evaluation taking place in worker processes, each expected run on a single compute node.

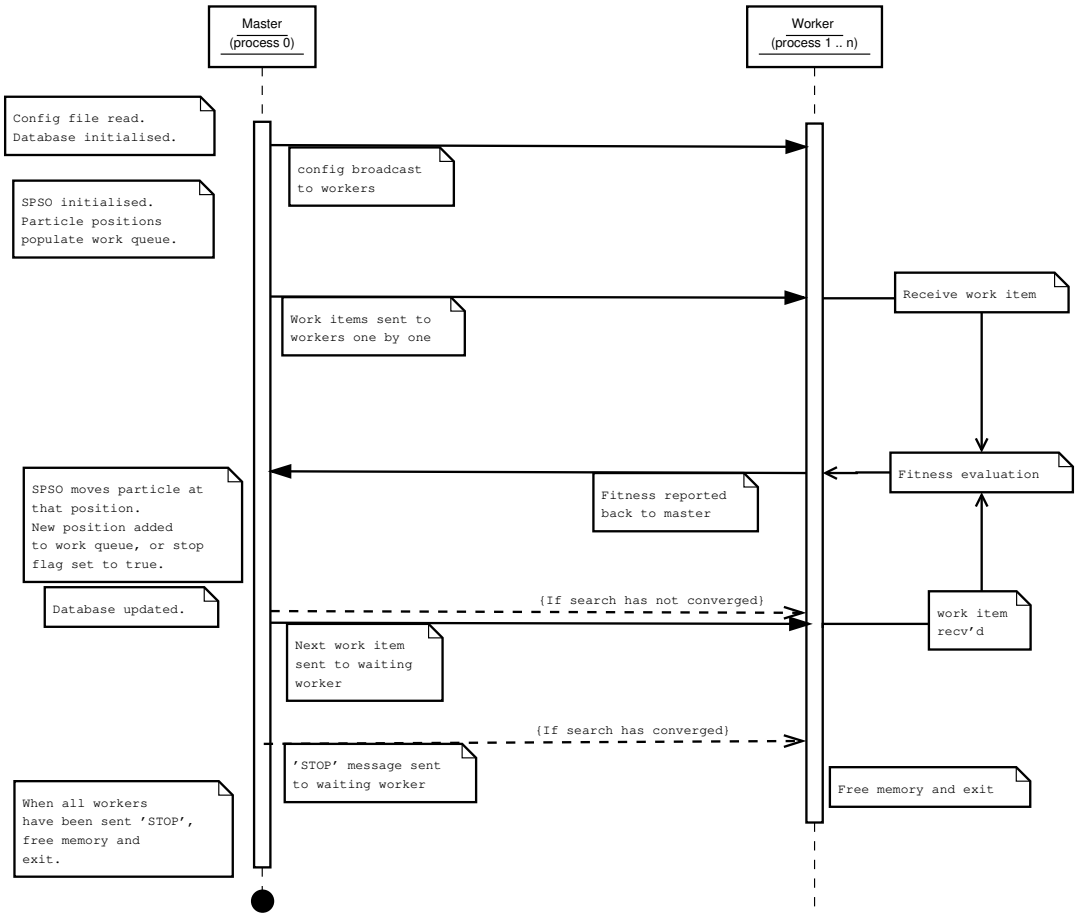

Figure 7-1: `OptSearch` *sequence diagram*

It should be noted that MPI processes are not expected to share nodes with other MPI processes, to avoid interference from their work affecting benchmark results. This cannot be reliably enforced in `OptSearch`, and is left to the invoking user to make sensible choices when structuring the job script.

7.2.1 Language choice

The final tuning application is implemented in C rather than Python. There are a number of reasons for this, already described in section 7.1.1.

7.2.2 Distribution of work

An MPI task farm is used to distribute work across a set of homogeneous nodes. The fitness tests do not depend on one another, so aside from updating the optimizer with the results, the problem could be considered almost embarrassingly parallel. This is partly enabled via the use of an asynchronous optimization algorithm.

7.2.3 Search algorithm

The search itself is guided by an asynchronous implementation of PSO, based on the SPSO described in [33]. This is used to feed the task farm queue, with the workers of the task farm supplying the fitness evaluation results back to the optimizer.

PSO is thought to be more tolerant of the type of search space likely to be encountered, and particularly of the compiler/compilation failures at points in it.

7.2.4 Categorisation of flags

After examining the manuals for GCC, the Intel and the Cray compilers, it became clear that there are three main types of flag used to control optimization options. The Cray compiler is very different from the others, and so was not targeted in this work.

Although all of these would be mapped to integer ranges so that they can be used with SPSO, dividing them in the configuration and in their initial handling allows for later work where they could be treated differently.

The three main types are on/off (`-fX`/`-fno-X` in GCC), list flags, which have a number of valid values to choose from, and range flags, which are similar, but the values are selected from a range of integers. These may take one of two forms with GCC: `--param X=Y` or `-fX=Y`.

Some flags affect other flags, but this was found to be a very small number in the case of GCC, and with this being the main target of this project, it was decided to avoid the complexity of handling these to save on implementation (and debugging) time.

The input file format chosen was YAML, as it is well supported and widely understood. It is more user-friendly than XML, yet still has the advantages of inheritance and grouping. Flag dependencies can be expressed, but are currently ignored by the parser.

7.2.5 Choice of optimization algorithm

It was clear from the beginning that the search space would be very large (even though most of the dimensions are short, there are more than 400 of them), and that the algorithm used to explore this space must be robust to compiler error. Prior experience of choosing suitable combinations of flags by hand had shown that compiler errors are to be expected, and that failure modes can range from an immediate failure (in one case, this was a floating point exception, raised by a flag combination found by prototype 1, much to the author's surprise at the time) to failure to terminate[3].

[3] Without waiting for eternity, it can't be verified that it would never terminate. The halting problem is definitely outside the scope of this thesis.

Through the early prototypes, it was made clear that the local minima occur *much* more frequently than was originally anticipated, and that a search algorithm must not only be robust but also likely to converge within a short enough time to make it viable for use in our high level use case (section 5.1). This experience led the author to realise that the search space described in [6] is very similar to that being explored in this project. When discussing their choice of optimization algorithm, Almagor et al helpfully explain:

> "To our knowledge, this paper is the first experimental characterization of these search spaces. Some of the results are intuitive; for example, these spaces are neither smooth nor continuous, so that simple steepest descent methods are not guaranteed to reach a global minimizer." [6, page 2]

One popular approach, used also by Ansel in the example from OpenTuner, is to use a genetic search in these cases. This was considered, but discarded in favour of PSO [49]. It was felt that this approach would be just as robust, but produce results more rapidly. There are also examples of both asynchronous (e.g. [65] and [124]) and parallel (for example, [151] and [168]) implementations, which have shown to be close in performance to the sequential, single-threaded original.

Within OpenTuner, a simplistic PSO implementation is included and this was used in later prototypes for this project. However, simplistic PSO does not perform well for high dimensional bounded search spaces, as described in [77]. In addition, the implementation within OpenTuner was not intended for tightly bounded integer search spaces. However, it was usable, and so useful as a prototype; it gave some hints on problems that might be encountered later.

The chosen implementation of PSO was based heavily on SPSO 2011 [33], with some small modifications to account for the difficulties of working in a discrete integer space. The velocity update function is the geometric function listed by Clerc in [33] and the algorithm has had some tuning, initially using parameter values from [134], which were then adjusted based on findings in early experiments.

The neighbourhood is global for all particles. This is the simplest method for updating the swarm when a good position is found; all particles are informed. While [31] suggests this may not be ideal in a continuous Real space, it appears to be less problematic when the search space is confined. This is an area that would benefit from further research.

Swarm size

The particle swarm size is calculated at runtime, and is one greater than the number of dimensions. This was a decision made after reviewing the literature. As explained by Clerc in [30], there is still a lack of research in this area, but it is clear that one must ensure the swarm size is large enough to avoid the search being limited to a subspace, while still being small enough to be practical.

In n dimensions, n points are bound to be in at most an $n-1$-dimensional space (e.g. in 3 dimensions, three points define a plane, or possibly only a line). Since in PSO, the particles tend to more towards or away from each other, if we have n (or fewer) points, the particles will tend to be in the $n-1$-dimensional subspace defined by the initial set of n points. It could be argued that $n+1$ points only define a simplex (generalised tetrahedron), whose volume will be small compared with the whole space, but fixing this problem fully would require 2^n points, which is infeasible. In practice $n+1$ has seemed satisfactory, though again more research would be useful.

With a swarm this size, it does not make sense to have more worker processes than there are particles. `OptSearch` would most likely not scale beyond the number of flags being searched as some workers would be starved of work. A small future improvement might be to use: $max(\text{number of MPI ranks}, n+1)$, however there were practical limits during testing and development that meant this could not be explored.

Boundary conditions

As explained in section 7.1.2, particles tend to leave the search space rapidly and are not guaranteed to re-enter it, so performance is highly dependent on the strategy employed at the boundaries and the velocity update function chosen. This is well documented by Clerc in [30]. There are also potential problems in getting stuck in a subspace or on a boundary [31], and these need to be avoided.

Dampening is still required, for the same reason as for the bouncing approach; it lessens the likelihood of leaving the search space repeatedly each time the position and velocity are corrected. In this design, the dampening factor is required to be large, and it is still possible that the particle will loop around the space for a significantly long time that it is necessary to intervene.

After some thought and mild experimentation, this dampening factor, d, was calculated thus:

Algorithm 2 Calculation of the dampening factor for particle movement

1: $d \leftarrow rand/(INT\_MAX + 0.1)$

Stopping criteria

It is important to know when to stop searching, as otherwise the search may continue on indefinitely without any improvement. This would be a waste of valuable compute time and benefit no one.

Since one cannot know in advance what answer is being sought, the usual practice in this type of search is to declare convergence when "not moving much".

In addition to being used to determine how much variation may be tolerated within a particular benchmark run, the value for `epsilon` (ε) in the configuration file supplied by the user to the tuner is used to determine whether the swarm is no longer moving very much.

In the case of our tuner, "not moving very much" is defined as being when improvements in the current global best fitness recorded for the current search are not larger than $\varepsilon \times \varepsilon$ for 200 iterations. 200 is a limit that was settled on after testing with this and higher limits as it seemed to give the best result without waiting too long. In practice this limit is encountered infrequently, with searches ending due to the limit in less than one fifth of tuning attempts.

Algorithm 3 describes the actions of the master process in deciding whether the search should stop.

Note that if the stop flag is true, then any further requests for work from workers are responded to with a value that indicates they are to exit cleanly.

7.2.6 Fitness evaluation

The obvious way to evaluate the fitness of a specific point in the search space is to time an appropriate benchmark. Provided the benchmark runs for a sufficiently long time, this should

Algorithm 3 Checking if the search has converged

```
Receive fitness (and position) from worker
if fitness < current best fitness then
    current best fitness ← fitness
    current best position ← position
end if
ω ← current best fitness × ε
if ω × ω > |(previous best fitness − current best fitness)| then
    if no-movement counter ≥ 200 then
        stop flag ← true
    else
        no-movement counter ← no movement counter + 1
    end if
end if
```

side-step the problems discussed in [176] that require such careful cache treatment and other steps to obtain an honest (representative of real codes) result.

For this to work, the benchmark chosen must be representative of a real code, and must run for long enough to be meaningful. As discussed in chapter 6, it is necessary to run the same performance test or benchmark, more than once. Otherwise it cannot be shown that a particular result is valid and neither a freak occurrence, nor subject to a higher degree of variation in performance than is acceptable. Further detail on the need to repeat the benchmark multiple times is given in section 7.2.7.

7.2.7 Repeated runs of the benchmark

As explained in section 7.1.3, it is necessary to run the benchmark a number of times. In this design, the fitness is determined by running accuracy tests first, and then, if those pass, the benchmark is run multiple times. This repetition of the benchmark should be configurable, to an extent, by the user.

Ideally, since there seems to be no established method for choosing the number, there would be a strategy chosen that is sensible. Several immediately present themselves:

- Best of n runs (n to be chosen by the user or set to a sensible default).
- Mean over n runs (n chosen by user or set to sensible default).
- Search for convergence over a maximum of n runs (n set by user or set to sensible default). This would require more input from the user so that a judgement can be made of whether or not convergence has been achieved.

The final choice made was a combination of all three. The user sets a threshold for the maximum number of runs, and supplies an expected deviation (standard error) from the mean.

The benchmark is run repeatedly. If n is less than 3, then the mean of n runs is returned. If n is equal to or greater than 3, the arithmetic mean and standard deviation are computed for the number of runs at each iteration.

In the early prototypes, validation of benchmark results was performed by curve fitting against $ax+b$ (using a least squares approach). This was computationally intensive, and, al-

though the required time was dwarfed by the runtime of the benchmark, as it had to happen multiple times per point in the search space, the additional time was significant.

Experimentation suggested that a highly simplified approach of comparing standard deviation of the results to a percentage of observed mean would suffice. This requires the user to specify the expected standard deviation as a percentage of arithmetic mean, but greatly reduces the amount of time required to assess the benchmark results for unacceptable variation and reject them if necessary.

In the final implementation, the user supplied value `epsilon` (ε) is treated as a percentage of the arithmetic mean, representative of the acceptable maximum value for standard deviation at any valid point in the search space.

In short, if over n runs, the benchmark results have a standard deviation greater than $\varepsilon \times$ arithmetic mean, it is assumed that the result is a failure as the performance varies too greatly.

In addition, each attempt to run the benchmark must terminate correctly (return value of 0) and within the period of time set by the user as an acceptable timeout for the benchmark.

Similar timeouts are given and used for the build and test cycles. If any overrun, or are deemed to terminate improperly (non-zero exit value), then the fitness assessment does not progress further.

If these conditions are satisfied, the arithmetic mean of the wall-clock time of the benchmark runs so far is returned as the fitness value of that point, and no further attempts to run the benchmark are made. This can save considerable time if each run of the benchmark is relatively long, as is usually the case if it is representative of user code. Note that at least 3 results from the benchmark are required for this assessment to be made. This can be overridden by the user.

In the failure case, the fitness value returned is `DBL_MAX`.

Algorithm 4 shows a simple approximation of the algorithm employed within the worker processes of the task farm. The master process performs all I/O and uses the search algorithm to populate the work queue.

7.2.8 Choice of PRNG

PSO has also been shown to be sensitive to the choice of PRNG algorithm [32]. This is hardly a surprise, as the PRNG is used to guide many of the updates during its run, as well as govern the choice of starting positions of the particles.

The vast majority of implementations of PSO available appear to rely on a Mersenne Twister [120]. Cryptographic randomness is not required, and often reproducibility is desirable, so this is not a bad choice. There have been some improvements since 1998 however, and so the PRNG chosen for this project was WELL512a [132][4]. This algorithm comes with a good test suite, and has received good reviews since it debuted in 2005.

With kind permission of the original author of the algorithm, the WELL512a code was modified slightly to supply integer values, rather than double precision floating point. This could be done without great risk (on current architectures), as the implementation uses `uint32_t` and the only required modification to the original is to avoid the conversion of the generated `uint32_t` value to `double`.

As the PRNG is used as part of the optimizer, which runs only in the (single-threaded) master MPI process, the problems of distributed or multi-threaded PRNGs are entirely avoided.

[4] On the advice of JHD, who knows more about these things than the author

Algorithm 4 A simplified description of the fitness evaluation performed by a worker process

```
1:  START
2:  Receive work item (string of flags) from master process
3:  if work item corresponds to STOP value then
4:      EXIT
5:  end if
6:  Run clean command
7:  if exitcode == 0 then
8:      Run build command, using flags to set FLAGS environment variable
9:      if exitcode == 0 then
10:         for i = 0; i < n; i++ do
11:             Run benchmark; record execution time {Note that the benchmark execution will be
                terminated if it overruns the user-supplied timeout. This is interpreted in the same
                manner as the failure exit code.}
12:             if exitcode! = 0 then
13:                 fitness ← DBL_MAX
14:                 Return fitness value to master process
15:                 Goto START
16:             else if i > 3 then
17:                 Compute arithmetic mean of recorded times so far
18:                 Compute standard deviation of times recorded so far
19:                 if standard deviation > (ε × mean) then
20:                     if i < n then
21:                         continue
22:                     else
23:                         fitness ← DBL_MAX {This point in the search space is too unstable}
24:                         Return fitness value to master process
25:                         Goto START
26:                     end if
27:                 else
28:                     fitness ← mean
29:                 end if
30:             end if
31:         end for
32:     else
33:         fitness ← DBL_MAX
34:     end if
35: else
36:     fitness ← DBL_MAX
37: end if
38: Return fitness value to master process
39: Goto START
```

7.3 Robustness

Any design for use in real world conditions needs to be robust in the face of a number of problems that cannot be avoided in software, such as compiler bugs and hardware errors.

7.3.1 Recording progress and resuming after termination

In the target use case, the software is expected to run under the control of a workload manager or scheduler, and so must also be able to exit cleanly when the allocated wall clock time comes to an end. It is highly desirable to be in a position to stop and restart the search using different numbers of nodes between each run, without adverse effects on the results of the search.

The design of `OptSearch` records the results of each fitness evaluation in a `SQLite` [82] database updated only by the master MPI process. `OptSearch`, if restarted using this database file, will resume the search from the last point at which a fitness result was recorded. It is not necessary to use the same number of MPI ranks each time, as only the task farm is affected by the number of available workers, and no part of the search depends on the number remaining the same throughout.

This is useful for systems administrators who may be using a strategy known as backfilling for their own work. While not truly elastic as `OptSearch` does not allow for changing the number of available MPI ranks while the search is running, it can be stopped and restarted many times with different numbers of nodes available each time.

The minimum number of MPI ranks required is 2.

7.3.2 Terminating cleanly

In order to end the search cleanly, it is necessary to do two things:

- Listen to and respond appropriately to process signals sent by the workload manager or scheduler daemon to the `OptSearch` processes.
- Have the master be in a position to inform workers that the search is stopping, whether due to converging or due to receiving a corresponding signal.

Signal handling functions are available as part of the Linux system libraries, so it is relatively straightforward to write something that should be portable across all Linux distributions (and most POSIX-compliant systems) and that can satisfy the first of these requirements.

The second is a little more difficult if the workload manager does not send the signal to all the MPI ranks. In such circumstances, it may be necessary to wait for each worker to finish their current work items before the processes can properly terminate. At present, the most commonly used workload managers, SLURM and PBS do signal all processes, and additionally allow the user to specify what signal they wish to send to the running program at when the wall clock time or walltime expires.

For `OptSearch`, the signal is set by the user in the configuration file, although to work around a bug in some versions of SLURM, another signal must be listened to regardless of what the user chooses, as it SLURM does not always heed this. This was the case on `Balena`.

7.3.3 Robustness against compiler bugs

Several problems were uncovered by prototyping. Firstly, there are some flags on which a compiler, say GCC, will simply error even though they appear in the documentation, the

source and the output of the various informational flags to the compiler (for example, `gcc --help=optimizers`). Secondly, it was rapidly apparent that some combinations of flags will cause the compiler to throw a variety of errors. One such combination caused GCC to throw a floating point exception while attempting to compile a simple "Hello, world" program. More commonly, the compiler will suffer a segmentation fault. Thirdly, some combinations may cause an unusable executable to be produced. This happens less commonly, and was in fact seen more often with the IBM compiler than with GCC, although it happened with all compilers experimented with.

Another serious problem that had to be worked around was flaws in the documentation. These were usually omissions. This applied not only to GCC but also to commercial compilers such as those from Intel.

In addition to checking the value of the return code when a command is executed, the following techniques were employed.

Execution timeouts

As was discovered early in prototyping, some combinations of compiler flags can cause the compiler to error in a variety of ways, but most can be grouped into one of two types:

1. Sudden failure (both segmentation faults and floating point exceptions have been seen)

2. Failure to terminate (deadlock or infinite loop)

It is also possible for the compiler to generate invalid code, resulting in one of the two types of outcomes given above (numerical errors and related problems are addressed in section 7.3.3 and section 7.3.3).

The first is not a problem, beyond the failure needing to be detected. Fortunately, on Linux and other *NIXes, the POSIX standard [142] has ensured that there is already a defined way to detect execution failure or success by examining the code passed on exit to the parent process. This is not guaranteed, as it requires software developers to adhere to the standard, but it should be sufficient for this use case.

Detecting a failure to terminate, either of the compiler or the compiled code, is more difficult. This requires a timeout to be set, the length of which should be set by the user in the configuration file. A different timeout is needed for the different phases of the fitness evaluation cycle.

Each worker is running a series of commands for cleaning up the build environment, the build or compilation itself, and then the accuracy and benchmark tests. Any one of these stages could fail to terminate, so a timeout needs to be enforced.

The benchmark is not expected to take the same amount of time to run as compilation, so at least two timeouts should be specified by the user in the configuration file and enforced by `OptSearch` during each fitness evaluation.

Any timeout triggered during the fitness evaluation cycle is treated as a failure, and the worker informs the master process by sending an appropriate value, `DBL_MAX`. If there is a failure in an earlier part of the fitness evaluation cycle, there is no point continuing with the evaluation and the worker will abort the cycle for that position in the search space, moving on to request the next work item.

Avoiding invalid or inappropriate results

As mentioned in section 2.1, performance can vary between processors that are thought to be identical (same manufacturer, part number, stepping, etc). Differences in performance can also be brought about through the operating system configuration, link order and seemingly innocuous features of the environment within which a benchmark is run [125], in addition to the benchmark itself, and the way it is timed [19] [176], particularly if the benchmark has a very short duration.

Aside from careful choice of the benchmark, it is vital to treat the results with care and rely upon statistical methods to determine their validity before using them to make performance comparisons and decisions.

Further work must be done by the operator prior to running the tuner, similar to that described in chapter 6. Where individual compute nodes differ greatly, either the value for ε (`epsilon` in the configuration file) must be large enough to reflect that, or the nodes used for tuning must be a subset of the available compute nodes that exhibit greater uniformity between them than observed across the complete set.

Too large a value for ε could result in an invalid result from the tuner.

Optimization correctness

Compiler optimizations, by their nature, require a certain amount of license in rewriting and translating the instructions of the programmer. This can be particularly problematic where numerical operations are involved, such as in the scientific software libraries that this project is targeting.

To ensure that software is not modified in a detrimental way, it is necessary to rely upon tests provided by the user of the tuner. These must be thorough and have full coverage of the software being tuned if they are to be useful.

The user of the tuner is also responsible for providing the set of compiler flags that comprise the search space. This cannot be done entirely blindly. Although it is common (for example, in the examples provided in [8]) to search over the entire set of flags produced by the union of the lists given by `gcc --help=optimizers` and `gcc --help=params`, it should be noted that this set includes *all flags that affect optimization* and almost entirely consists of machine-independent optimizations. The result of this is that several compiler flags that are intended for instrumentation, along with flags that alter the way numbers are treated, are included in this set: Their use has consequences for the optimization passes of the compiler, potentially altering the semantics of the generated code, in addition to the resulting speed of the executable. The set of flags that GCC provides when `gcc --help=target` consists of machine-dependent optimization flags, but it does not make sense to search through all of these. They are more clearly a mixture of optimizations that may not always be appropriate, with some applying to TLS instructions or the generation of 32 or 64-bit code (in the case of x86), while others enable or disable different widths of SIMD instructions and are clearly of interest when tuning many types of scientific code. As with the other flags, it would appear that these must also be pruned to a more sensible subset before any meaningful search can be carried out.

Some compilers, notably GCC, do not object to conflicting flags being passed to them on the command line. It is usually stated within the documentation that either the flag listed first or that listed last overrides all others. They also disable certain instrumentation flags at some pre-defined optimization levels (such as `-O2`) that are enabled by default at lower levels. In

the case of GCC, one such flag is `-fwrapv`, which is disabled at `-O2` and higher.

It is possible that a developer may be relying on certain kinds of behaviour from the compiler, wisely or not, and that these may be affected by the choice of compiler flags used. This is not something that the tuner of this project can be expected to deal with, and it is necessary for the user to take care in choosing the flags to search over, and to have familiarity with the software being tuned. The aforementioned `-fwrapv`, for example, changes the way that signed integer overflow is defined, which may have serious consequences depending on what behaviour the programmer has assumed.

Compensating for poor choice in input is extremely difficult, and outside the scope of this project.

This leaves us with another question: Is it possible to surpass the work of those who blindly search over all optimization-affecting flags, who do not make any attempt to check correctness[5] of the resulting binaries or that the benchmarks are providing a reliable measure of performance? This seems unlikely, though it could be argued that any results gathered under such circumstances are of questionable utility. Any checking would have to be performed by the user after tuning is conducted, which presents difficulties in compensating for any problems that might be discovered.

A more interesting question, which is dealt with later in this work, is whether or not it is possible, with the described restrictions imposed on the search, to best the flags chosen by the compiler developers to correspond to `-O3` (or the more conservative `-O2`)?

7.3.4 Fault tolerance

As with any large and complicated piece of machinery, supercomputers suffer the occasional hardware or software failure. The most common types of failures differ by machine, its cost, the limitations placed on its administrators and the experience of the users and type of usage.

The most common types of non-hardware failure seen tend to be either network congestion, often caused by badly written software, or nodes going down due to Out Of Memory (OOM) conditions. The former can sometimes be remedied by user education, and its impact lessened by good interconnect design and tuning. There are mechanisms for dealing with the latter, but these are not always implemented for a variety of reasons; lack of knowledge or time on the part of the administrators, or limitations imposed by local politics, being the biggest. This tends to be a problem only on smaller, less well resourced systems, especially those with a high rate of change of users (and unexpected, less typical usage) as is seen at universities.

The damage caused by temporary loss of individual nodes is generally limited to those jobs running on those nodes. This is relevant to a distributed tuner because it is possible that, if a bug is discovered during the compilation stage, a node may be lost and that fitness evaluation result (which should be reported to the tuner as a failure) will be lost also.

It could be argued that OOM should trigger the Linux `oom-killer` if the nodes are correctly configured. Without doing a large-scale survey of HPC systems, it is impossible to know whether it is unreasonable to assume sensible behaviour of the `oom-killer`. Such a survey is outside the scope of this project. Fortunately, it is also not required. Linux allows the setting of various resource limits on child processes via the `setrlimit()` system call, and GNU has extended this with `prlimit()`. Similarly, time limits can be set on the execution length of such processes, which should help offset the problem of a compiler bug that results in a deadlock or infinite loop being triggered by a particular choice of flags.

[5] in the sense of compilation

Hardware errors are a fact of life on a large machine. Components have known failure rates, so administrators can approximate the number of failures to expect during any particular month or year. Higher quality, and thus more expensive, components generally have lower rates of failure, so it is the users of lower-end machines who bear the brunt of the problems. Similarly, new and emerging hardware is liable to errors in design or manufacture that may make soft errors more likely.

These failures cannot be predicted before the errors are reported, and on production systems there will be set thresholds for replacement of each type of component. Only when these thresholds are met are components replaced, as to do otherwise is both impractical and unaffordable, especially on very large systems.

Any software running on these systems must therefore be able to cope with a small number of soft errors occurring in components such as DIMMs. More serious errors are likely to be unrecoverable, making checkpointing or a similar method of storing and restoring state such that any processing can restart with minimal loss of progress highly desirable.

7.3.5 Non-homogeneity

The working assumption of this project is that nodes within a supercomputer or cluster are homogeneous. Unfortunately, some soft errors, such as processor bugs that cause the CPU to be stuck in a particular P-state, can seriously undermine this assumption. A distributed tuner could easily discard the "correct" result because its performance was affected by such a bug.

Fortunately, these problems occur infrequently, once a system has been baselined and a homogeneous set of nodes selected. However, in future it is likely that this process will become more difficult, as it can be seen that recent errata published by chip manufacturers, particularly Intel, have continued to grow in size, with individual issues increasing in severity from the point of view of the programmer or HPC centre manager. The recent specification update for Intel ® Skylake [93], for example, lists several instances where a processor may suddenly suffer from very slow execution of AVX instructions, or become stuck in a particular P-state.

In such circumstances there is no warning, and there is no operating system support available to detect these types of errors. Most systems administrators are limited to waiting for users to alert them to the sudden occurrence of unexplained performance issues.

The old method of timing by cycle counting is not accurate enough in these cases, as the numbers of cycles are affected and hardware counters have been observed to count too many or not at all. This makes it worse than useless, and unlikely to be representative of what a user might see. Timing by wall-clock time is crude, but, provided the benchmark is long enough, should avoid the worst of the issues while maintaining some semblance of reality.

7.4 Conclusion

The process of prototyping has highlighted a number of practical challenges that will have to be addressed in the final design. The first is the choice of a language that will be well supported on emerging hardware: The final design will be implemented in C, the language used for the most commonly used operating system in supercomputing. The workload will be distributed across multiple nodes of the supercomputer being targeted using a simple task farm.

After experimenting with several alternatives, the optimization algorithm chosen for this project will be SPSO [33], using periodic boundary conditions to deal with particles moving beyond the edges of the search space. This will be supported by WELL512a [132], as SPSO

is highly sensitive to the PRNG used with it. The swarm size will be set to one greater than the number of dimensions being searched (a future improvement of $max(\text{MPI ranks}, n+1)$ has been suggested).

The benchmark being used for tuning of the targeted library will be run at least 3 times, up to a maximum set by the user, with convergence being judged to have happened if the results do not differ by more than 2 standard deviations. That is, they will be within 1 standard deviation of mean. If this does not happen, the fitness evaluation result will be deemed invalid as the variation at that point in the search space is too high.

Results from each fitness evaluation within the search space will be recorded in a SQLite database, such that any completed valuations are not lost when the tuning application terminates before the search has converged. The tuner, when restarted, will initialise the SPSO particles from the stored data, so that the search may continue from the last recorded point.

It must be assumed that the user of the tuner will know to provide a valid set of tests for the library being tuned, and to provide a benchmark that exercises at least those parts of the library that are expected to be most heavily used on the target hardware. Checking such things computationally is outside the scope of this project.

It is also clear that to fully meet the requirements, a number of assumptions have to be made about the target machine and the way in which that machine is managed. These have been set out in this chapter, and draw heavily on the author's own experience in managing similar systems. The main two assumptions are that the nodes used for tuning are homogeneous, and that the system administrators have already checked for soft errors and other problems that may compromise the homogeneity of these nodes. These nodes are expected to be representative of the compute region of the supercomputer where the applications using the tuned libraries will be run by users.

CHAPTER 8

DESIGN IMPLEMENTATION

Unlike the early prototypes, the final implementation does not read directly from GCC's output or parse its source, nor does it assume any particular compiler or target language. Instead it relies upon the user to specify all flags and options for the compiler for the search. Due to time constraints and the difficulties of doing such things in C, there is minimal error checking on the inputs, so the user must ensure that the compiler flags supplied are valid before proceeding. Failure to do so is likely to result in the search converging very rapidly with a less than useful result, as the various attempts at recompilation fail to show any improvement in performance.

For ease of use, on those systems where Python has suitable support, a script has been written to programmatically determine these when the target compiler is GCC. This script will also attempt a binary search for the limits of any valid values where such flags are of the type that requires an integer value within a fixed range. For GCC, one file from the source is required, `params.def` (for GCC 4.9.3 this is found within the source tree at `gcc-4.9.2/gcc/params.def`). This supplies, where available (not all are set), the maximum and minimum or set of values, along with a default for each parameter (flags that take the form `--param foo=X`). For other flags, the output of the appropriate section of the help, such as `--help=optimizers` or `--help=target`, is used. The author intends to reimplement this in C in future, to avoid the portability problems that have been seen on emerging architectures when using Python.

Unfortunately compilers other than GCC, particularly commercial compilers, tend not to be so easy to query. There is no standard way to provide this information programmatically, so it is necessary to construct the input file by hand, which can be a tedious and time-consuming process.

The `OptSearch` application is started using `mpirun` or `mpiexec`, or whichever mechanism is in use for starting MPI programs on the target machine. I/O, other than debug information, is handled in rank 0, which is designated the master MPI process. This process reads in the configuration file, and performs queries and operations on the SQLite database. This database is created if it does not exist, and contains a record of every position the PSO swarm has visited and its fitness score, if one has been determined. This is in addition to storing the current best position and several other parameters required to restart the search with the swarm initialised to the positions the particles were last in.

As shown in the high level sequence diagram given in figure 7-1 in section 7.2, the worker processes, ranks $1..n$, wait for instructions. These processes make up the workers of the task

farm, and do little more than the fitness evaluation cycle, the result of which is passed back to the master process and stored in the SQLite database. When it is determined that the search has converged, or an appropriate signal is received, the workers will stop any running child processes, receive the STOP signal from the master process (since it is not always possible for all processes to be informed through other means), and exit cleanly. The master exits after all workers have been told to stop work, and the database has been updated with the latest results.

For increased robustness, the SQLite database uses the Write-Ahead Logging (WAL) journal mode [83], with auto-checkpointing and the "synchronous" flag set to 3 or "extra". This has allowed the database to survive even a fatal segmentation fault from within SQLite itself (see section 8.5.2).

8.1 Component break-down and encapsulation

To aid in debugging, unit testing and to facilitate rapid development, the various parts of the software is divided into separate areas, each with its own API. This allows for some parts of the software to be worked on while others are waiting for attention, and may be "stubbed out", or implemented to a very minimal level. Each component can be tested in isolation, which has proven instrumental in finding and fixing bugs (and verifying that they are, and remain, fixed).

The dependency graph shown in figure 8-1 gives some indication of how this has been done. The task farm, for example, is encapsulated within the file `taskfarm.c`, with its accompanying header file defining the API through which the other parts of `OptSearch` may interact with it. This approach has been utilised throughout the design.

In the call graph (figure 8-2), it can be seen that most of the implementation details are found within `optimizer.c`. This makes a number of calls to the APIs defined for each of the components of the software, as might be expected, but there are few calls between the components.

Some components, those with functionality provided by third-parties, such as the PRNG, are hidden from the rest of the application by small wrapper programs (e.g. `random.c`). This encapsulation is intended to allow the rapid replacement of such components if a better implementation is found later. It also allows unit tests to remain untouched if such a change is made.

8.1.1 Unit testing

Each part of the application was designed so that it could be tested in isolation, or replaced without tests needing to be rewritten to remain effective.

Very simple unit tests have been used to ensure the functionality of each component remains unchanged after refactoring and bug-fixing. Each time a feature has been added, a simple test has be added at the same time. Equally, when bugs are uncovered, a test has been created to reproduce that bug, so that it can be ensured that bugs are fixed and are not re-introduced. Through this simple, well-established (in software engineering circles) process, considerable time and effort has been saved during each stage of development.

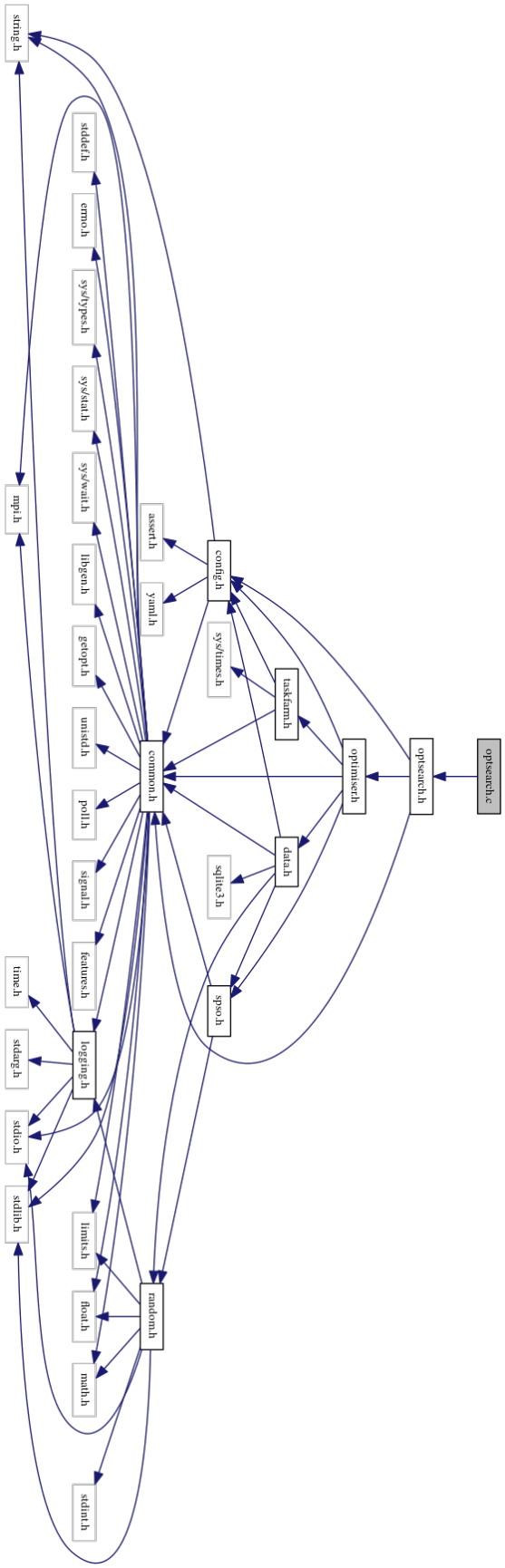

Figure 8-1: *`OptSearch` dependency graph*

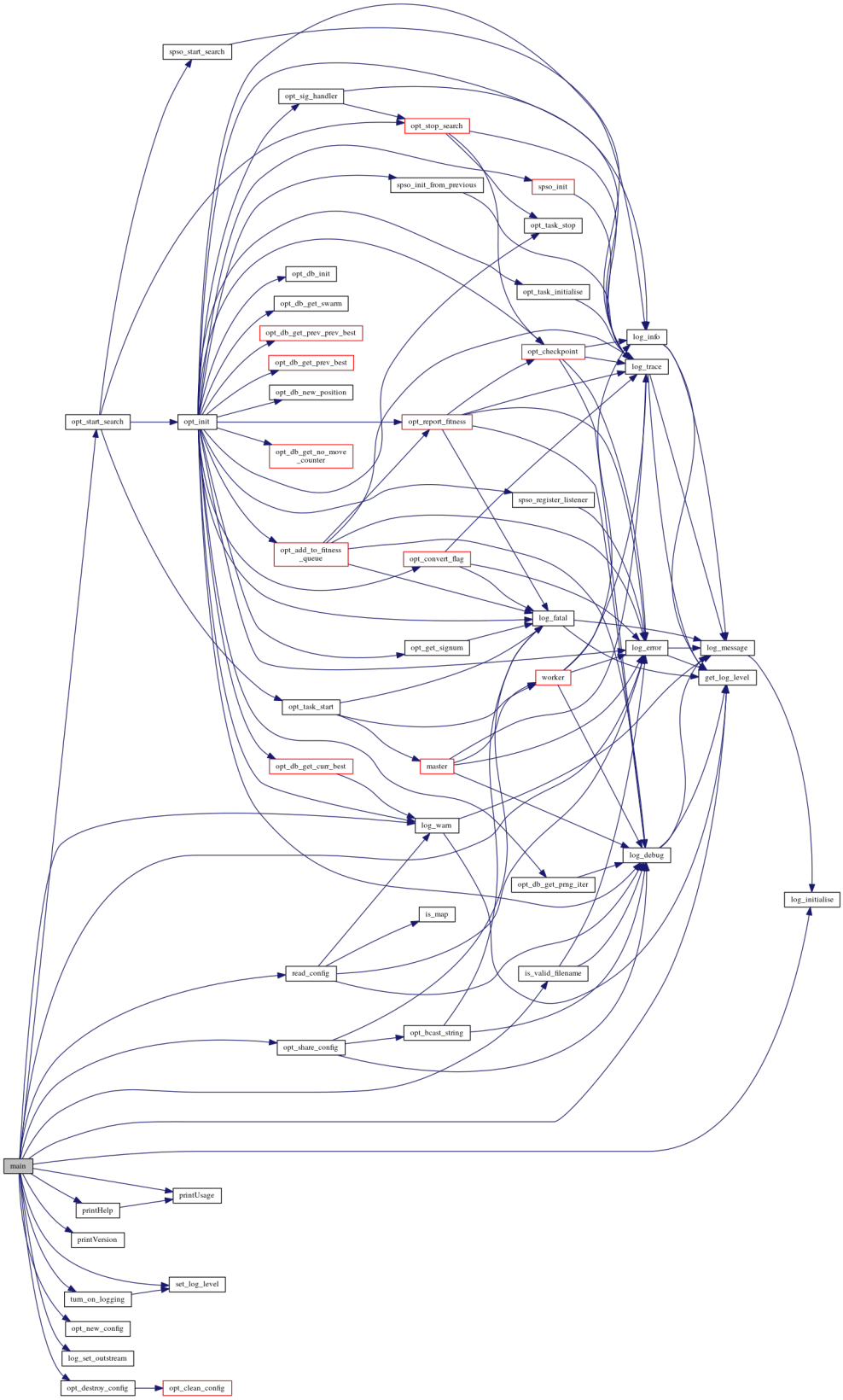

Figure 8-2: *`OptSearch` call graph*

8.2 Alterations to target software

As might be hoped, few alterations are required to be made to the target software in order for it to be tuned. Those made for the experiments detailed in chapter 6 and chapter 9 are explained here. Most were made in order to simplify the process of automating the clean,build,test,benchmark cycle.

1. To avoid building the benchmark every time, dynamic linking was used. This required mildly altering the build system of the reference BLAS to ensure a shared library was created.
2. By default, the LAPACK build system, which also builds and runs the BLAS tests, will rebuild its own copy of the BLAS library and test that. This had to be altered so that the test programs would link against and use the shared library created in 1.
3. The benchmark, HPL, was built to link dynamically against the shared BLAS library. This benchmark was not rebuilt at each search point, as it is only optimization of the BLAS library that should be being tested.

To avoid introducing a bias into the results, all comparisons to default optimization levels are based on data obtained using exactly the same software and wrapper scripts (wrapper scripts are not a requirement to use this software, but were used for convenience).

8.3 Wrapper scripts

As is the established practice on many systems, software on `Balena` is provided through the use of modules [63]. While this approach is extremely useful in allowing several versions of the same software to co-exist and be switched between by users at whim, it presents a problem when executing commands that depend on particular software versions being available. It is difficult to ensure that the environment is retained, and it is also possible that the benchmark may be built differently, e.g. using a different MPI implementation, to `OptSearch`.

The simplest workaround to this problem is to enclose each command in a wrapper script, which sets up the environment prior to issuing the command of interest. A word of caution: The wrapper script *must* preserve the return code of the command it is wrapping around. Without this, the tuner cannot determine whether or not a particular command has executed successfully.

Unfortunately, this means that, when execution is timed, the script is timed too, and any overheads incurred are counted as part of the fitness value of the benchmark. For this reason, the wrapper scripts must be kept as simple and clean as possible. Those used for this project introduced a consistent overhead of 0.8 seconds, which is far shorter than the runtime of even the fastest execution of the benchmark.

In the case of HPL, this approach (of timing the entire runtime) is interesting, since it will always give a slightly larger (in the order of a few seconds) result than is reported by HPL itself, regardless of whether a wrapper script is used or not. This is because HPL times only part of its execution, the additional set up time varying slightly depending on the machine on which it is running. That said, provided that the overheads are small, constant and apply to every benchmark run consistently, this should not prevent the tuner from making its assessment with enough accuracy to produce a useful result. In this case, although the compilation options do affect this part of the benchmark, the overall runtime of the HPL is substantially longer and the impact of compilation choices on the set up/tear down sections is extremely small by

comparison (around 0.5s of variation at most, versus hundreds of seconds for the benchmark itself), thus having little influence on the tuner.

8.4 Signal handling

All processes listen for signals telling them to exit cleanly. This was less simple to implement than expected, due to a problem in the version of SLURM on `Balena`, and other quirks of the operating system. Some of the quirks that apply to OpenMPI do not apply to MVAPICH, and vice versa. In order to be MPI-implementation agnostic (assuming that OpenMPI is usable with applications of this type in future, given the problems discussed in section 8.5.1), is is necessary to implement workarounds that will work in all of these situations.

- `SIGUSR1` will usually kill a bash script
- `SIGSTP` and `SIGCONT` signals are consumed by OpenMPI's `mpirun`, unless the appropriate MCA parameter is supplied

The MCA parameter required for signals to be properly forwarded by OpenMPI is `--mca orte_forward_job_control 1`.

In addition to these issues, there are signal-handling quirks that depend on the scheduler or workload manager in use on the machine being tuned on. On `Balena`, SLURM was found to be accepting but ignoring the `--signal` directive. This is thought to be due to a combination of the SLURM version and the way it has been configured. To work around this, it was necessary to hard-code listening for two signals that were always sent by SLURM when the job was either cancelled or the walltime limit was reached. Those signals are `SIGINT` and `SIGCONT`. The `SIGINT` signal is also useful during testing, since the application then behaves as expected when the user issues a `Ctrl-C`.

8.5 Difficulties encountered in third-party software

While Python-based prototypes had many problems (see section 7.1.1), implementing the final version in C was not without issue. In part this was due to the extra functionality in this version, and the difficulties in writing and debugging parallel applications. However, some difficulties were caused by issues with third party libraries.

Despite efforts to keep the number of third-party libraries used by `OptSearch` to a minimum to aid portability, it was necessary to make use of the work of others for MPI and for the database used to store information about the search.

8.5.1 Overly-helpful libraries

Early on in development, OpenMPI [161] was used on `Balena`, but this ran into difficulties. `OptSearch` must call `vfork()` followed immediately by `execve()`, to start a child process for each stage of the fitness evaluation cycle (clean, build, test, benchmark) in the worker MPI ranks. It must also record the exit code of the child process each time, while ensuring that the child process does not overrun the user-defined timeout.

OpenMPI uses `posix_atfork()` to register a `fork()` handler and thus detect the call to `vfork()` (or `fork()` if that is used instead). If detected, OpenMPI will not allow the code to continue, issuing an error message. Later versions of OpenMPI can be made to issue

a warning instead [162], but, if the child process also uses OpenMPI, the `exec()` fails due to the way memory is used by OpenMPI in both processes.

Switching to a simpler MPI implementation, MVAPICH [71], solved the problems and allowed `OptSearch` to run correctly regardless of whether the invoked process (HPL in this case) was using MVAPICH, OpenMPI or something else.

The appeal of OpenMPI is the extent to which its performance can be changed at runtime depending on the selection of the parameters passed to it through parameters, environment variables or config files. The amount of effort that the developers have gone to is clear, but they also have very specific use cases in mind, better suited to codes that are not spawning child processes.

OpenMPI does a great deal of checking for mistakes and misuses of the library by its users. OpenMPI actively detects the spawning of child processes, and the comments from the developers of the library imply that they consider this to be a case of user stupidity (or old code), rather than an unusual yet legitimate use case [153]. It seems likely, given the memory corruption that results when trying to use child processes within OpenMPI, that design decisions have been made early in the development process for this library that do not allow for this type of usage.

8.5.2 Bugs in SQLite

A bug in SQLite caused some jobs to be curtailed. This appears to be a segmentation fault caused by invalid memory access within the query parser, particularly within the query 'walker'. Unfortunately there was not time available to track it down further once it became clear that producing a fix would be very time consuming. The tuner was able to resume cleanly once restarted, especially since the queries that caused problems were always those that checked whether a position had already been visited. Due to the number of columns (one per dimension), this led to a very long query, although executed by hand and, most of the time, by the tuner, it did not cause any errors. It was encountered whether or not SQLite was compiled with thread-safety turned on, something that should have been unnecessary with only one MPI rank (itself single-threaded) performing all SQL and file I/O operations.

Fortunately, with the database configured to be as resilient as possible, data is not lost due to this and jobs merely terminate prematurely. The frequency of occurrence is high enough to be irritating, with approximately 10% of each tuner invocation (via job submission) ending in this way. Simply resubmitting the job is enough for the search to resume where it had stopped, and so in practice the author was able to work around this by automating re-submitting. The user need only check the output (or exit code) to determine whether the short runtime means that the search has already converged or if the SQLite bug has reappeared. No changes are required to the input to re-submit.

8.6 Conclusion

The final design was broken down into components, so that it could be both implemented and tested piecewise. This and comprehensive unit tests allowed implementation to proceed more rapidly, and with less difficulty, that it might otherwise have done.

Some changes were made to the target software to allow it to be tuned within a shorter time frame, necessary where local policy places limits on the available computation time. This was achieved by avoiding unnecessary repeating of build steps of the benchmark and test exe-

cutables, which would not themselves be targeted for tuning. These would normally use static linking and thus require at least re-linking with the library for every fitness evaluation within the search space.

To achieve this, the Makefile of the `BLAS` library was adjusted to build a shared rather than static library. Similarly, the Makefile building the `LAPACK` tests was altered to link the resulting executables dynamically with a shared library, and the `HPL` benchmark was similarly altered to link dynamically against the shared `BLAS` library. This was not necessary for `OptSearch` to work, but saved considerable time in the tuning cycle.

For the invocation of the build, test and benchmarking steps, short, simple wrapper scripts were written to facilitate use of the `module` [63] system. This provides access to software on many modern supercomputers, including `Balena`. These scripts were kept as simple as possible to minimise overheads, which were checked to be both very small (a few milliseconds) and, so far as possible, non-variable.

It should be noted that even when wrapper scripts are not used, `OptSearch` will still show a runtime longer than that reported by `HPL` by a few seconds, because `HPL` reports the time for only a part of its execution (omitting, for example, the allocation of memory for the matrices or the reading of its input file). In contrast, `OptSearch` will time the entire execution period, which has been seen to be as much as 4 seconds longer on `Balena`, and longer on systems with slower memory access. Without making the tuner benchmark-specific (and so target-specific), it is not possible to avoid this, but so long as these set up and tear down times are consistent, it should not make any difference to the efficacy of the tuning.

Unfortunately, some problems with third-party software were encountered that caused failures. These were easily worked around due to the robust, standards-compliant design of `OptSearch`, but their diagnosis was time consuming.

One was a design decision by the authors of OpenMPI not to support calls to `fork()` or `vfork()`. This was a relatively minor problem given that `OptSearch` could be built against any MPI implementation compliant with the MPI-3.1 standard

The second, an obscure bug in SQLite, will need to be tracked down and reported to the SQLite project team in the near future. Fortunately the design of `OptSearch`, in being robust to premature termination, is able to work around this problem. It can simply resume from the last recorded step in the search. Using the WAL journal mode, along with autocheckpointing and increasing the "synchronous" setting of SQLite to 3, helps to support this.

A third problem was encountered in SLURM, the workload manager on `Balena`. `OptSearch` will always terminate cleanly when it receives a `SIGINT` or `SIGCONT` signal, in addition to whatever signal the user requests is used within the `OptSearch` configuration file. This ensures correct behaviour at the end of a job on any system using the SLURM workload manager, where some versions ignore the `--signal` parameter.

CHAPTER 9

RESULTS

Most results were obtained from Bath's `Balena` (appendix A.2.6). The first attempted search (that did not end in a failure due to human error) targeted GCC 7.3.0. This was the latest version at the time, and was built with all the required libraries to enable all possible optimizations, especially those for loops. As stated previously, the compilation target was the reference BLAS [117], using HPL [136] to provide a fitness evaluation and LAPACK [112] to check for correctness of compilation.

Restrictions in the scheduling policy on `Balena` meant that the job was limited to 4 nodes for each run, and could run for no longer than 6 hours. Later, access was granted temporarily to a higher priority queue that allowed the experiment to be repeated on larger numbers of nodes. This made it possible to verify that the tuner successfully scales at least to 16 nodes on `Balena`, and that this substantially shortens the time taken to complete the search, changing it from around 100 minutes to 20-30 minutes for the 255 flags used in section 9.1.2. It is believed that the tuner will scale substantially beyond 16 nodes, but further testing on one machine seemed of limited utility with this test case. A longer-running benchmark might benefit more from greater numbers of nodes being utilised.

The same scripts for building, testing (via the BLAS tests included with LAPACK [112]), benchmarking and cleaning up between each run of the experiment were used for all experiments. The only change was to the flags passed to the compiler in each case, and the environment was ensured to be identical every time aside from these flags.

Results were obtained using HPL with N=9600, NB=100, P=Q=4 (WR00R2R2). The exact `HPL.dat` can be found in appendix C. This problem size was chosen after initial experimentation to discover the largest problem that could deliver meaningful results while still being practical within the restrictions imposed on users of `Balena` and within the time available for this project. For the larger problem size referred to later, only N was changed.

9.1 Current Intel®IvyBridge

As we saw in the study in chapter 6, which used the same scripts and benchmark (and problem size), there is a lot of variation between groups of nodes within `Balena`. For the purposes of this section, a sub-set of nodes was chosen to ensure true homogeneity; all had the same memory type and processor stepping. These nodes were the 80 nodes within the partition "batch-128gb", which are fitted with 128GB RAM (DDR3 1866 MHz) in single-rank DIMMS,

Number of flags	Best result	Mean	Mean time to convergence	Number of samples[1]
254	87.4s	92.2s	00:32:28	3

Table 9.1: *Results for GCC 7.3.0, searching over machine-independent flags and parameters only*

without including the first 22 nodes (delivered early in the machine's life) that were fitted with memory by a different manufacturer (Micron). That left nodes `node-sw-[023-080]`; 58 nodes in all.

The results of benchmarking these nodes also provided a useful figure for `epsilon` so that the tuner, `optsearch`, could discard any points in the search space exhibiting unusually high variation in performance. That is, results with a standard deviation higher than `epsilon%` of mean at that point in the search space (on that node).

The tables in the following sections should be compared to the figures in table 9.4, which includes baseline figures for `-O3` and other compiler-defined optimization levels.

9.1.1 Results without -O flags

The tuner was first configured using lists of flags from the GCC 7.3.0 manual, and from `gcc --help=optimizers` combined with `gcc --help=params`. This did not include any architecture-specific flags, as the first list includes only machine-independent optimizations. Some of these optimizations, having been deemed unsafe for scientific libraries, were omitted from the configuration even though they may affect performance.

This list was pruned further when it was found that many flags that increase the search space considerably (having valid values within a large range) are not required for performance. This reduced the search time and, as an unanticipated side-effect, reduced the number of local minima encountered by SPSO.

The second list consists of flags, all of which take the form `--param name=value`, that control various parts of the compiler's optimization machinery. They make it more or less effective depending on the targeted code and architecture.

Results are shown in table 9.1 and were obtained using 4 nodes (64 cores; one MPI rank per core). Although only three data points are presented, further results that were obtained during development suggest this behaviour is typical and that it is not greatly influenced by choice of PSO parameters.

Note that all of the results in these tables are reported times from `OptSearch` itself, rather than from HPL. This means they include the entire runtime of HPL, not just the section that is usually timed by the benchmark.

9.1.2 Results with -O flag

The tuner configuration from section 9.1.1 was re-used, with the addition of the `-O` flag, which takes values `0` through `3`, `fast` and `s`. This produced a faster result, while not increasing the search space greatly. As before, it was found that a more aggressive pruning of the flags supplied to `optsearch` resulted in convergence on a better result.

Results are shown in table 9.2, and were obtained using both 4 and 8 nodes. One of the results used 4 nodes initially and then 8 nodes when re-started. Increasing the number of MPI ranks did not reliably shorten time to convergence.

[1] Number of times `OptSearch` could be used to search for the best flag combination in the time allowed.

Number of flags	Best result	Mean	Mean time to convergence	Number of samples
255	12.3s	19.8s	01:26:59	4

Table 9.2: *Results for GCC 7.3.0, searching over machine-independent flags and parameters, with the addition of the* `-O` *flag*

Number of flags	Best result	Mean	Mean time to convergence	Number of samples
431	94.4s	96.4s	02:06:08	2

Table 9.3: *Results for GCC 7.3.0, searching over both machine-independent and -dependent flags and parameters, with the addition of the* `-O` *flag*

The best result found used 175 flags (see appendix B), of which 39 required a value to be supplied, 33 of which were numeric values chosen from within a large range.

9.1.3 Results with -O and machine-dependent flags

The configuration from section 9.1.2 was augmented to include a sub-set of the flags produced by `gcc --help=target`. Not all of these flags are relevant to dense linear algebra, and some apply to older processors than IvyBridge. As it extended the required time for searching, there was also a desire to keep the search space as small as possible. Most of those flags chosen were related to SIMD instructions. Results are shown in table 9.3. 4 nodes (64 MPI ranks) were used.

9.1.4 Other techniques

The existing methods of obtaining a higher performing library are demonstrated here. The most obvious, using one someone has already written, is possible now that Intel®IvyBridge is a well-established microarchitecture. Results are given for ATLAS [178], and it should be noted that it took more than 2 working days and almost as many CPU hours to produce the library that was used for this.

The next most obvious method involves simply choosing the right `-O` and letting the compiler work it out, or taking the time to look through the compiler manual and attempting to choose the best flags, depending on the library, target architecture and compiler being used.

Table 9.4 uses the reported runtime (in seconds) from HPL. `OptSearch`'s best result is included for comparison (note that the result here will look shorter, as it is the performance reported by HPL, rather than the total wall clock time). It should also be noted that ATLAS gains a small advantage by being statically linked, but this cannot account for the scale of the difference in performance seen.

[1]...and several years of experience

[2]Once the library author has gained access to the hardware, and had time to add support for it

Method	Mean runtime (s)	Standard deviation	Number of samples	Library build time (approximate)
`-O0`	95.40s	0.06s	56	<1 hour
`-O1`	26.74s	0.02s	174	<1 hour
`-O2`	17.10s	0.02s	180	<1 hour
`-O3`	13.08s	0.03s	177	<1 hour
`-Ofast`	13.06s	0.04s	175	<1 hour
Hand-picked flags	10.03s	0.03s	60	1-2 hours[1]
Best from `OptSearch`	9.93s	0.04s	6	1-2 hours
ATLAS	2.99s	0.01s	241	50 hours[2]

Table 9.4: *Results for `-O` flags, ATLAS, and hand-picked flags*

9.2 Intel®Ivybridge as it would have been

As explained in section 4.3, the target for this work is not established, well-supported architectures. This means that the numbers thus far supplied are not necessarily representative of the results that might be obtained if the tuner is used in the intended situation: That of having access to a machine for which there are not yet any high-performance libraries.

In order to attempt to replicate something of the intended target situation, the same experiment was repeated with an earlier version of the compiler, GCC 4.4.7. This version was supplied with the operating system when the machine was installed, and its change log shows that no support specific to IvyBridge had yet been added. Results are shown in table 9.5, with the search space consisting of both machine dependent and independent flags, with the addition of `-O`. This is not quite equivalent to the experiment that produced the result in table 9.2 as a greater number of flags was included in the search and there are several flags which have either been removed from or added to the set for the later version of the compiler. This makes it impossible to compose an equivalent search space.

This result of 21.3 seconds, which HPL reports as 19.48s, is not impressive, because using `-O3` alone produces a runtime reported by HPL as 13.78s. Unfortunately, due to time limitations, it was not possible to investigate whether a smaller search space might have resulted in a better result as it did for GCC 7.3.0.

It should be noted that IvyBridge, being of the x86 family, is not as unsupported as the ARM chips mentioned in section 4.3. Despite lacking support for the microarchitecture, GCC 4.4.7 produces reasonably optimised code due to its similarities with other Intel family chips.

Number of flags	Best result	Mean	Mean time to convergence	Number of samples
289	21.3s	36.5s	01:06:22	6

Table 9.5: *Results for GCC 4.4.7, searching over machine-independent flags and parameters, and a subset of machine-dependent flags, with the addition of the `-O` flag*

9.3 Discussion

Despite the only difference being the addition of the `-O` flag, adding only one dimension to the search (complicated in that, depending on the value taken, this flag activates or deactivates several flags), the results in table 9.1 and table 9.2 are surprisingly different. The addition of this "hyperflag", if it may be so-called, appears to have had the effect of lifting the search out of the local minimum it has previously become stuck within, preventing it from becoming prematurely stuck. In early experiments, before settling on what seemed to be the best parameters for PSO in this search space, it was already clear that PSO could be highly temperamental, at times appearing to alter its behaviour depending on the ordering of the dimensions.

It is interesting to see that the hand-picked flags and the best result found by `OptSearch` deliver very similar performance, yet on examination, use very different sets of flags. A crude count of the number of generated instructions with references to `xmm` and `ymm` registers, to give some indication of the extent of vectorisation, immediately shows that the hand-tuned binary is more highly vectorised. Looking only at `DGEMM`, which is the only part of the BLAS that is really exercised by HPL, the hand-picked flags generate 513 SIMD instructions referencing both types of register, 233 of them referring to `ymm` registers used in `AVX` instructions. In contrast, `-O3` generates no `AVX` instructions and only 140 SIMD instructions that refer to `xmm` registers, while `OptSearch`'s best results generate around 859 SIMD instructions, all of which refer to `xmm` registers only.

In addition to these numbers, table 9.6 includes the second best point found by `OptSearch`, which is comparable in performance with the hand-picked flags (mean runtime of 10.1s). This is particularly interesting as this provides 3 unique points within the search space that give similar results for this problem size. Two of these points, both of those found by `OptSearch`, do not make use of the widest vector instructions. This is true even when the problem size is increased to 12800, where `OptSearch` manages to deliver a mean runtime of 130s, vs the hand-picked flags' runtime of 128s (for GCC 7.3.0).

In spite of the differences in vector instructions, all three of these (the two results from `OptSearch` and the result from hand-picking the flags) rely on loop optimization flags and produce results noticeably better than `-O3` alone for both N=9600 and N=12800. Even so, it is clear from table 9.4 on page 103 that it is not time for Whaley, author of ATLAS, to retire yet. Whether hand-tuning of flag choices is used, or automated tuning with an approach such as that described here, the gap between the unoptimized performance of the reference BLAS and ATLAS can only be narrowed by 30% at most.

Larger problems than N=12800 could not be investigated due to constraints on job runtimes and the need for a single fitness evaluation to complete in order to record its result. It is not possible to checkpoint the benchmark during its run, and interrupting the compilation and testing phases is not possible without substantial further work.

Flags	XMM instructions:	YMM instructions:	total SIMD instructions:
`-O3`	140	0	140
`-O3 -mfpmath=sse -mavx`	174	23	192
Hand-picked	285	233	513
`OptSearch` 2nd best	398	0	389
`OptSearch` best	859	0	859

Table 9.6: *Comparison of numbers of SIMD instructions generated by GCC 7.3.0 when compiling DGEMM*

9.3.1 Reproducibility

An additional experiment was run, this time with the compiler flags in the input file mildly shuffled, to see whether the result obtained previously could be reproduced independently of the ordering of the search dimensions. This initially failed to run to completion.

In this case, the search seemed to get stuck after approximately 7 hours, with particles bouncing between a subset of positions in the search space. In total, 1573 visits had been made to 532 unique positions within the search space. This is a tiny amount given its size and the time spent searching.

With the options again shuffled slightly, and a different allocation of nodes (access having been granted to a higher priority queue), the experiment was re-run.

The behaviour was little changed whether the job requested 2, 3, 4 or 8 nodes. On examining the SPSO algorithm in more detail, it was determined that, in a discrete search space such as this, the geometric algorithm used to compute a new velocity and position for each particle was not moving the particles efficiently, and they are instead inclined to revisit the same point repeatedly. A small modification was made, using a threshold for the number of visits to each position in the search space. When this is exceeded, the particle will instead be moved to a random position and given a random velocity with a probability of 50%. Initially, the threshold was set to a static number (200 visits). This solved the problem of SPSO fixating on one position, but did not show much of a progression through the search space. Instead, the additional random jumps ensured that the search was more likely to accidentally chance upon the "right" answer. The result is clear in the `global_best_history` table in SQLite, where the current global best answer is at `DBL_MAX` until it jumps to the final result and stops after SPSO decides that the search has converged (see table 9.7 for an example of this).

After some thought, and discussion with James Davenport, The threshold was altered to be dynamic, its value dependent on the number of positions already visited. It is calculated using a suggestion from James (see algorithm 5 on page 105).

Algorithm 5 Calculation of the threshold for movement to a random position (and velocity) in the search space. This happens 50% of the time that this threshold is reached.

1: **if** times this position has been visited $<2 \times$ max(1, times this position has been visited $\div$total visits to all positions) **then**
2: Move particle to a random position in the search space
3: **end if**

With this alteration, the behaviour of SPSO was much improved, showing a steady progression towards the best answer, and avoiding becoming stuck. This can be seen in table 9.8 and the corresponding figure 9-1 (note that "search iteration" is not strictly valid, as the search is not iterative but driven by results returned by the task farm).

Perturbing the compiler options within the input file twice more and re-running the experiment showed `OptSearch` finding the 'best' result of approximately 10.0s in 50% of tuning

timestamp	positionID	fitness	visits
2018-05-30 12:52:13	6	`DBL_MAX`	2
2018-05-30 12:49:14	512	131.601115	3

Table 9.7: *Table showing search progression for one search on `Balena`, where only two points are visited.*

timestamp	positionID	fitness	visits
2018-05-30 18:09:53	3	DBL_MAX	48
2018-05-30 18:12:55	98	26.296122	5
2018-05-30 18:12:55	99	23.292912	17
2018-05-30 18:15:26	124	12.28193	221

Table 9.8: *Table showing search progression for one search on* `Balena`

attempts (the worst result found was 33s), although the time to find it varied from 16 minutes to 1 hour 10 minutes on 16 nodes. This is thought to be due to the random nature of asynchronous SPSO and the distribution of 'good' results within the search space.

While usually very similar, the chosen "best" flags were not identical in each case. Some flags, such as `--param predictable-branch-outcome=15` might take a different value, such as 21, in one set of found flags, while others, such as `-fstack-reuse=named_vars` appear in only one result. Aside from these few, the vast majority are identical, suggesting that the few that deviate have little influence on the performance obtained.

Interestingly, increasing the search space to the full list of possible flags, as might be done by a naive user, appeared to have a detrimental effect. `OptSearch` returned to finding "good" points in only 5-15% of attempts to search for them. Perturbing the flag list did not appear to make any difference, although this is not conclusive as not many experiments of this nature were run. Those that were suggest that PSO may be not only highly variable and potentially affected by the flag ordering, but that this variability may increase as the search space increases.

It is worth noting that [33, Figure 3.1] considered at most 44 dimensions, and says "there is still a lack of theoretical analysis" for the optimal swarm size, never mind other SPSO parameters. In this thesis, up to 250 dimensions have been typically considered, with best results at 175. Hence it is not clear whether the difficulties encountered with a greater number of flags are generic to SPSO, or a characteristic of our particular problem. The reader is referred to Clerc's comment about a lack of theoretical analysis.

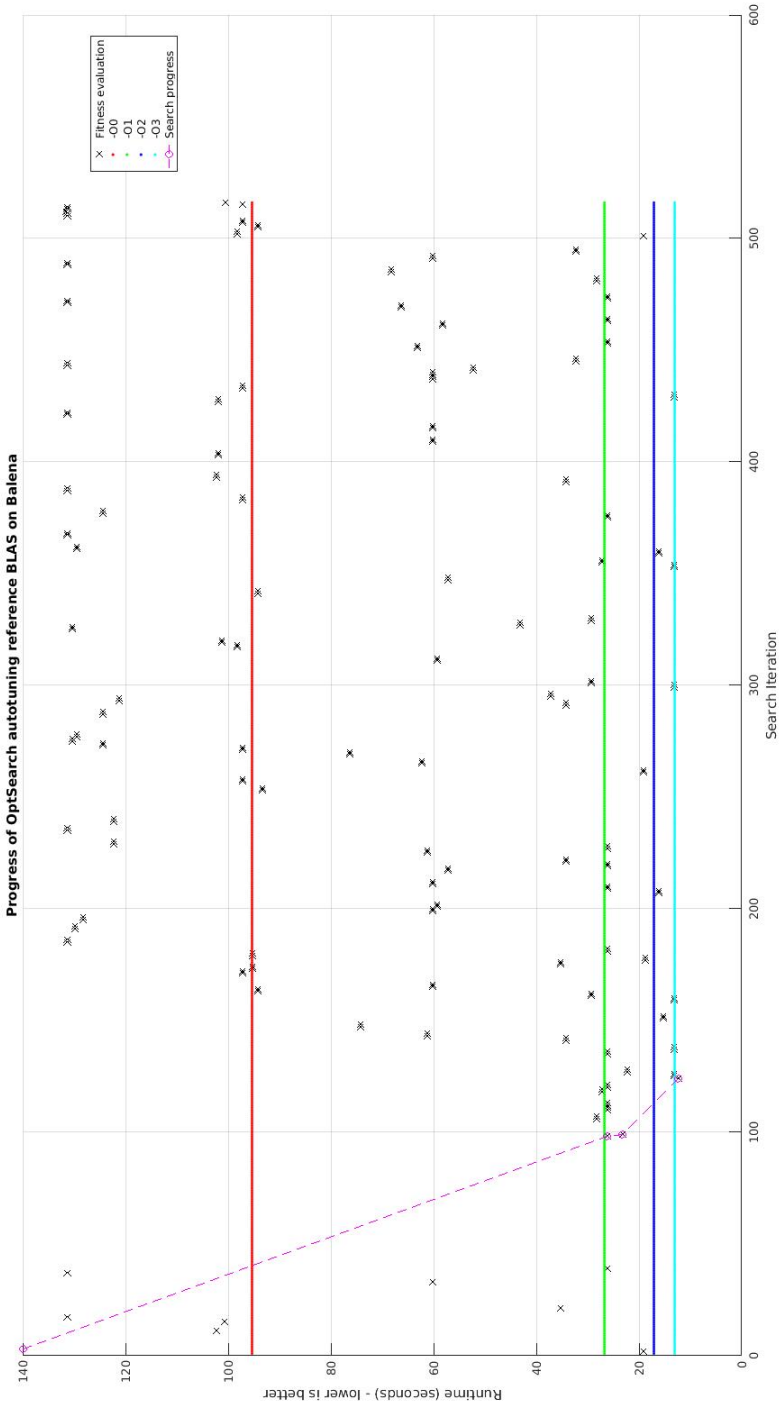

Figure 9-1: *Progress of one search on* `Balena`*, showing striation of performance at the various search positions*

CHAPTER 10 EVALUATION

As could be seen in table 9.4 on page 103, the commonly used `-O3` can be improved upon in a way that is significant: Using `OptSearch` performance improved by 24%.

This is consistent with the best improvements found in [6] with their own compiler. Unlike `OptSearch`, Almagor et al do not restrict the search to avoid numerically unstable or highly variable results. Although the compiler is their own and not GCC, it performs similar optimizations.

Gains such as those achieved by a purpose-designed and built library such as ATLAS cannot be achieved. However, this optimization can be done by someone with limited knowledge of the target library, provided that adequate tests are provided, and unlike ATLAS has the potential to deliver performance gains with other libraries than the BLAS.

Unfortunately, the approach chosen is not without its problems. Most of these appear to be a result of the nature of the search space and its effect on the search algorithm being used. This is perhaps unsurprising, given the complexity of this type of optimization [28].

10.1 Observations of compiler performance

Despite the high expectations of improvements when moving from GCC 4.4.7 to 7.3.0, `-O3` resulted in an average runtime for HPL (as reported by that benchmark) moving from 13.78s to 13.08s, an improvement of only 5%.

Large hardware manufacturers, such as Intel, are known to contribute to compilers (and others, such as GNU binutils, libraries such as libxsmm [76], and the Linux kernel [50]) as much as a year in advance of a new microarchitecture appearing on the market. A simple look at the change log in many projects makes this clear.

In binutils [66] 2.30, a project that includes the gold linker and various other tools required for the building of software on many, mainly open source, platforms, email addresses of contributors give an idea of the scale of these contributions. From 2000 onwards, of 1499 entries, 40 are from intel.com email addresses. This may not sound like many, but is huge compared to the 11 from AMD, 4 from ARM and 4 from Samsung. Others are from Linux distribution maintainers, a number of enthusiastic open source developers, and here and there a contribution from those who may also be interested in specific language support (adacore.com appears 3 or 4 times).

A similar distribution is seen in the GCC change log. It is thought likely that other chip manufacturers are less able to prepare the ground in such ways, as they cannot match the level of investment of the larger companies.

It is very disappointing that the improvements in performance are so small, despite this investment.

10.2 Observations of the search space

The progress of the search within the space shows obvious striations, as might be expected. Larger jumps in performance are thought to be due to turning on/off significant optimizations, while smaller variations around each point are due to tuning that optimization. This can be seen in figure 9-1 on page 107.

The tables of positions in the search space and their fitness values, generated during the search, show that the majority of points visited are failures, and many that are not failures return a fitness that is somewhere between the performance of `-O0` (no optimization) and `-O1`. This is consistent with what was observed during prototyping, described in section 5.2.2: The search space is very similar to that described in [6], in which the authors explain that, for their compiler:

> "80% of the local minima in the space are within 5 to 10% of the optimal solution."

Although their search space is significantly smaller, the author believes the search space for all compilers will be very similar, due to the way that compiler optimizations are performed.

10.3 Behaviour of search algorithm

The SPSO algorithm is very variable, as also observed in [14], who saw PSO finding a "good" result in only approximately 15% of searches. This is consistent with what was seen during `OptSearch` development.

With later improvements, the "best" result was found only in 50% of searches. Less if the search space was left at its largest (as it might be if a novice is using `OptSearch` to tune a library they know little about).

The dimensionality of this search space is far higher than is usually used with this search algorithm. For example, in [77] 40 dimensions is considered a large number, whereas, even ignoring machine-dependent flags, in tuning the options of GCC 7.3.0 there are 413 options to consider; 292 for GCC 4.4.7[1].

The initial search cut down (by hand) the number of options to search to 254 for GCC 7.3.0 (288 for 4.4.7) on `Balena`. Many are instrumentation flags irrelevant to the search, while some others are unsafe for numerical code.

In addition, SPSO is usually used to search within a continuous space in $\mathbb{R}$. When tuning compiler options, many of the dimensions are very short (within the interval $[-1,1]$), and all (in the case of GCC) are within $\mathbb{Z}$.

With these differences, it was difficult to know whether this search method would be the best approach. SPSO is not guaranteed to find the best answer, and the asynchronous version used in this project is not guaranteed to take the same path to the "answer" (or one of them,

[1] This is the RedHat packaged GCC and it appears to be missing some optional components, so the figure may be higher if built with all options

if there are several) each time. In practice, it often takes a similar, though not identical, path, presumably due to the homogeneous architecture and resulting similar times to compile, test and benchmark the target library.

10.4 Environmental considerations

This project began as a small hobby interest, and although it became a more serious research undertaking in 2007, has suffered from the difficulties of being limited in scale due to the availability of hardware combined with the policies chosen by those overseeing the university HPC resources. This has made it impossible to do more than plan for and guess at how well it might scale in practice.

In addition to these problems, the nature of part-time study and the lack of understanding or support at UK institutions for those studying in this way has made it increasingly difficult to make progress of any kind. Part-time students are liable to be overtaken by their full-time peers, who can donate more time to their work and tend not to have many commitments outside of their studies. This project had to be suspended several times, often for many months on each occasion, due to life events. In retrospect this is not a mode of study that the author would recommend to anyone. Younger students studying full-time can easily begin their PhD project later and finish sooner, rendering one's work obsolete in the process. Maintaining motivation in the face of such enthusiastic competition and university policies that seek to punish anyone who is not progressing "fast enough" is a constant struggle, even with extremely supportive supervisors.

CHAPTER 11 FURTHER WORK

While tuning the compilation process in this way has been shown to be of value, there are clear gains that can be made by improving the implementation. This is especially true where the search algorithm is concerned, now that a better understanding of the nature of the search space has been obtained.

11.1 Investigate behaviour of `OptSearch` at scale

Only very limited opportunities to investigate the behaviour of the tuner with larger numbers of nodes presented themselves. The limit on `Balena` was a mere 16 nodes, which could not give a good indication of how much faster the search might progress if more worker processes are available. In practice, we would expect this auto-tuning work to be carried out by the system staff during the early days of the machine's deployment, when the majority of compute nodes are available.

One avenue for further work would be to investigate the potential time saving that may be had on a very new machine where the system administrator could make use of more of the compute nodes. If this approach shows potential, that could open up further avenues for improvements to the software.

11.2 Other compilers

Obvious areas for expansion are to extend this work to other compilers than GCC. While GCC is currently the dominant compiler in this area, it is not the only compiler. Further improvements might be possible with one of its competitors, such as `clang` or `flang` from the LLVM project. These appear to be gaining in popularity due to their more modular architecture, and have been used as the basis for several commercially available compilers.

11.3 Other architectures

At the start of this project, it had been expected that some results could be obtained on other architectures than x86, but unfortunately access to these systems was lost and could not be reinstated in time to provide any meaningful results. ARM presents a particularly interesting

target, as it appears to be slowly gaining in popularity in HPC and has not been subject to the same level of care and attention from compiler and library authors as x86. Such emerging architectures (in the HPC space) were always the real target of this work, and it is hoped that they might benefit more than the incumbent architecture, x86, does.

Indeed, early work on `Amber` (appendix A.2.4) indicated that the gains might be substantially higher than for x86, emphasized by the poor performance of even auto-tuning libraries (no others were available) such as ATLAS at the time that it was brought into service. At that time, the tuning of parameterised kernels done by ATLAS, which took around 48 hours to complete, could result in achieving only around 20% of R_{peak}. In contrast, hand-selection of the GCC compiler flags for the reference BLAS was able to deliver performance that was very slightly better than this. (More if the compilation of the benchmark was tuned also.)

11.4 Other libraries

Targeting other libraries in the list of Dwarves [34] was always an intended extension of this work. Dense linear algebra may not be the best test subject, due to the amount of work that has gone into producing good results for the TOP500 [122] list.

It is anticipated that greater gains may be had where libraries have not been so heavily targeted for optimization by compiler writers, whose software now recognises the GEMM, especially DGEMM, and optimizes it accordingly.

For this reason, sparse linear algebra (eg Sparse Basic Linear Algebra Subprograms (SBLAS) [48] and ScaLAPACK [18]), and graph algorithms (particularly in light of the recent focus on the Graph500 [70]) may present a more interesting target.

11.5 From libraries to applications

There is nothing in the design of `OptSearch` that would prevent its use for applications or benchmarks. Only further experimentation can tell what the correct balance is between tuning libraries and tuning applications, but by targeting libraries, one can at least know that a broad range of applications will benefit, and to know what those applications may be before choosing a target library.

It could be argued, as [17] do, that one should target applications. One problem here is that applications such as DL_POLY in fact have many computational kernels, for different user calculations, and it is not obvious that the same choice of compiler options will work for a range of different user calculations. However, it might be possible to take a group of users using the application for similar calculations and produce an optimised version *for them*, with no guarantee that it was optimised for other uses. Thus, one may have to supply multiple versions of an application, each optimised for one of the most common uses of it; an approach that is probably not practical for an academic machine where use varies considerably and is apt to change during the typical lifetime of the machine.

Artificial benchmarks are a different matter. During acceptance testing of `Balena` (the Bath supercomputer listed in appendix A.2.6), the author was asked about improvements to the results of the HPL benchmark [136]. Initially, the results were around 20% lower than the acceptance tests required. The missing 20% was recovered by choosing more appropriate compiler flags for the benchmark itself, an approach the administrators had not considered initially.

This was in addition to the use of a tuned library, and for a benchmarking exercise, very significant (in this case, the machine could not have passed acceptance without it). Auto-tuning a benchmark (as well as the libraries on which it depends) has obvious advantages if one is competing for a position in a list such as the TOP500, but, as discussed in section 3.1, could be considered controversial.

11.6 Exploration of the search space

A study of the search space of widely used compilers, such as GCC, would be useful as it could inform work on improved search algorithms. Such a study would likely require the search to be performed on a subspace, perhaps selected to be representative of the search space as a whole, in order to be feasible.

11.7 Behaviour of PSO in similar search spaces

The behaviour of PSO, as discussed in section 10.3, has been shown to be highly variable. Further research into search spaces of a similar nature (via an appropriate objective function) would be beneficial, as it could inform work on further improvements to the PSO algorithm.

11.8 Alternative search algorithms

Since SPSO does not perform so well in high dimensionality spaces with short dimensions, another algorithm, at least as robust, should be sought. Before beginning this project, it was thought that the search might be approached as a graph problem, but it was quickly found that most compilers, and certainly GCC, do not have enough dependencies between flags to make this approach viable. (Or if they do, they do not document it.)

Further improvements may be gained by dividing the search space up into separate areas, where a machine learning approach could be used. Dividing the search space in this way may also provide a better method for attacking larger problems, which will be necessary as the numbers of flags for each compiler increases. In addition, this could provide a possible method for better scaling on future (exascale) systems.

Supervised learning approaches were originally deemed unsuitable by the author due to the additional complexity in ensuring that the final tuner produced by the project would be both portable and easy to use. The constraints on implementation time were also a significant factor in deciding against this type of approach. Since that decision was made, others have made progress using a variety of machine learning techniques [11].

As it will not be known how many “good” answers may exist, an algorithm that does not require this knowledge at the start would be best. One possible approach might be to use Dirichlet processes [55], if the distribution is known to be appropriate (this would require an exploration of the search space, as suggested in section 11.6) and there is suitable training data. A limitation of this type of approach is that it requires data from hand-picking flags. Availability of training data may make it impractical if this data does not already exist or cannot be easily generated. Additionally, reducing the number of dimensions required to search is likely to be beneficial regardless of the search algorithm chosen.

Another, simpler approach, which may work with GCC and similar compilers, might be to use a cost function for combinations of flags and parameters. This could be problematic

given the likelihood of encountering local minima. Such an approach would probably work best when combined with hill climbing, or another appropriate method, to find the best values (and combinations) of flags that take a value from within a (large) range.

11.9 Improvements for high-end supercomputers

On some high-end systems, particularly those within commercial environments, compilation can take place only on the login and service nodes, the compute nodes being reserved, as the name implies, for pure computation. (The choice of whether or not to allow compilation on compute nodes varies depending on the decisions made by those responsible for that particular system.)

This presents a problem for the current model of working, and due to the limits on CPU cycle usage on many login nodes (to try to enforce more friendly behaviour between the many users) it may be necessary to be careful about how compute-intensive the optimizer is. A multi-threaded approach could be highly desirable, with batches of compilations running at any one time (numbers concurrently running to be chosen by the user) with separate directories for all files generated. Then the fitness function would need to be capable of dispatching of a job to the work load manager, and then waiting for it to return with the fitness for that point in the search space.

This is likely to be very slow, and, with the intended audience being the system administrators, it may be worth considering cycle-stealing approaches. However, most systems administrators are likely to be capable of formulating a strategy using the workload manager on their systems that fits in with the local workflow. As there is much variation between supercomputing centres, such tasks are probably best left largely to them.

CHAPTER 12 CONCLUSION

This project has demonstrated that it is possible to tune the options of the most commonly used compiler to increase the performance of a commonly used library, in this case the reference BLAS, by 24%. This closes the gap between the performance such a library usually delivers when compiled simply (with `-O3`) and that provided by using a special-purpose high performance library, such as ATLAS [178]. It is also almost 5 times the 5% used by Biscof et al [17] to justify the comparatively high investment of a performance expert's time.

12.1 Answers to Questions

In addition, this project has provided answers to the key questions raised in section 1.2, given below.

RQ1 How necessary is machine code?

If ultimate performance, or anything close to it, is to be achieved, then machine code is still necessary for dense linear algebra. Even with a well-developed compiler (GCC 7.3.0), we have only been able to obtain 10.0s by hand-tuning the compiler's flags, still significantly slower than ATLAS's 2.9 seconds. However, it should be noted that it was more than a year before ATLAS reached this level of performance even on Intel®Ivy Bridge. This process often takes significantly longer on other architectures, such as the APM X-Gene 1 machine `Amber` (see appendix A.2.4) mentioned in section 4.3.

RQ2 Is performance-portable code achievable?

While it is not possible to match the performance of a carefully hand-crafted library, it is clearly possible to close the gap between the performance of these and the mediocre performance of the default optimization levels provided by the compiler by auto-tuning the choice of compiler options. Further, it can be done with little specialist knowledge, within an acceptable amount of time (<2 hours in the example used in this project).

The maintenance overheads of a high performance library, such as ATLAS or GotoBLAS [68] (or its descendent, OpenBLAS [180]), are much higher than for naive code. As explained in section 1.3, it takes some time, often in the order of several months, for the authors of libraries such as these to catch up with the release of new hardware architectures.

Naive code is easier to maintain and to debug. It takes less time for a developer, newly introduced to the code, to get to the level of familiarity required to maintain or expand it. These are highly desirable features of any software project that is likely to have a long lifespan. It is perhaps worth reminding the reader at this point that the BLAS first debuted in 1979, and, though more subprograms may have since been added to it, it shows no sign of declining in popularity.

RQ3 What is the economic value?

The lifetime of the typical supercomputer is now about 5 years, depending on the chosen refresh cycle of its supercomputing centre. If a high performance version of a library cannot be obtained for the first 1-2 years, that is a significant portion of the life of that machine.

Increasing the speed of computation of general purpose libraries used by the commonly run programs on that supercomputer will have a noticeable, and *cumulative*, effect on throughput over that period. This project shows that, for a small (1-2 hours on 4 nodes) cost in compute time, a significant improvement in performance can be obtained that would increase throughput for those applications that might otherwise be dependent on the unoptimized BLAS library.

Using the results given in chapter 9, and an estimate of the proportion (27.5%) of top applications that might benefit (using the figures given in [7]), we can adjust the calculation from [17, page 3] accordingly:

$$\% \text{ performance improvement} \times \% \text{ of applications using BLAS} \times \text{TCO of the machine} = 0.24 \times 0.275 \times \text{TCO of the machine} = 0.066 \times \text{TCO}$$

Using the total cost of ownership (TCO) figure for RWTH Aachen calculated by Bischof et al, this comes to around 363,000€[1] *for tuning only the BLAS library*.

This very naive calculation assumes that the tuning itself has no cost. For a single library it is very small relative to the lifetime of the machine. Full workings are given in appendix F on page 136, where the calculation is also given in CPU hours.

Even if the cost in CPU-hours is negligeable (which it is), there is also the cost in person-hours, which depends critically on the person's skill level. [17] assumed 2 person months per application and a 5% improvement: This thesis proposes a methodology which is almost certainly less expensive than a full tuning and delivers a 24% improvement, so is certainly more cost-effective than the already impressive method of [17].

12.2 Implications for Dense Linear Algebra

The community measures HPC machines by their HPL figure. Even for dense linear algebra, the relevance of this is being increasingly questioned (see, for example, [47, slide 16]), and the High Performance Conjugate Gradients (HPCG) benchmark [45] [46] is proposed as a replacement. Whatever the merits of HPCG (and the tuning of HPCG would require more attention to MPI parameters etc.), it is likely that HPL figures will be quoted for many years [47, slide 20].

[1] 6 person-years by their costings!

12.3 Implications for wider HPC

Bischof et al justified the high investment of hand-optimizing codes with only a 5% improvement in performance. This project has demonstrated that a greater improvement is possible at a far lower cost and shorter lead time.

It is expected that this work will translate to other libraries, such as FFT and sparse BLAS, although it has not yet been tested. It has also been designed to be largely language (and compiler) agnostic. In contrast ATLAS can only provide a BLAS implementation and support for compilers is largely limited to the GCC [174].

If this type of approach is used more widely, and authors of scientific software are careful in their choice of libraries, it should be possible to make emerging hardware more widely accessible to the scientific community. The end-users of these systems could benefit by avoiding the costly, non-portable performance tuning that they have so far been engaged in and that cause them so many difficulties when migrating from one system to another, and yet still be assured of reasonable performance early in the machine's production lifespan.

12.3.1 Suggestions for Application Authors

Authors of scientific applications should expect their code to last beyond the time they are likely to be using or working on it. Careful documentation, the use of well documented and established libraries, along with suitable tests for routines in those libraries that are critical to the code, would help those who come after them and also allow their code to benefit from auto-tuned libraries.

If the most performance-critical part of the code is not a call to a library, for a sufficiently complicated application it may be worth considering making a separate library for such routines. That way a naive, easy to maintain version can be created and kept separately from any version with machine-specific optimizations. This would provide a target for auto-tuning by `OptSearch`, and is likely to be helpful when testing (and if it is to be tuned, it must have suitable tests provided for it).

12.3.2 Suggestions for Compiler Writers

It is, perhaps, surprising that GCC 7.3.0, which is capable of generating AVX instructions, doesn't do so even with -O3. However, as discussed in section 2.6.2, this is understandable in light of the developers' main use case. This use case is not obvious to the casual user, so the advice to authors of compilers is to add some information about it in their documentation.

Further improvements to documentation, and more reliable ways to query the available flags, are recommended if users are to be in a position to choose appropriate flags for their target software. It would also aid approaches similar to that taken in this work.

12.3.3 Suggestions for Library Developers

Projects of this type would be assisted by libraries that provide simply written reference versions, in addition to a well defined API, as is the case with the BLAS. The provision of suitable tests, so that numerical stability can be checked, would help not only projects such as `OptSearch`, but those writing high performance versions of libraries in assembler. It would also benefit the users of such libraries, who could check that their chosen implementation is sound.

12.4 Implications for universities

In light of the problems hinted at in section 10.4, universities should evaluate their policies with regards to part-time students. These policies, usually written with full-time students in mind and then applied to both, tend to be unnecessarily hostile towards the part-time student. This is almost certainly due to a lack of understanding, as part-time study is uncommon enough that few who have experienced it are likely to be making policy regarding it. Policies that take into account the many challenges of part-time study, combined with appropriately designed support, would make the journey a much less stressful one and increase the likelihood of success.

APPENDIX A

INFRASTRUCTURE

It is worth considering that this work cannot have escaped influence from the environment in which it was conducted. Therefore, it is necessary to describe that environment, which itself changed during the course of the work. Nevertheless, the environment has always been considered to be typical of that provided to researchers at academic institutions, and not so dissimilar to those elsewhere.

Also noteworthy is the prevalence of machines based on the x86_64 architecture, all of Intel's design and manufacture. This is not an ideal target, as it is well established, and it is well known that Intel ensure the success of (the majority[1]) of their chips by preparing the ground before they arrive. This is done by contributing to, for example, the GCC collection and the Linux kernel, ensuring that the hardware is supported from the moment it goes on sale. Before, in fact. Such behaviours ensure that reasonable, if not good, performance can be expected on all their chips from the beginning, although the high performance libraries, including the BLAS implementations, still take time to catch up.

This level of preparation is not something many can afford to do, and gives a significant advantage to Intel over their competitors. Ideally, in a project of this type, experiments would be run on the "real" target of the work: An architecture that is so new that support for it is limited.

A.1 Software

The GNU Compiler Collection (GCC) has been the primary target of this work in part because of its widespread availability. This has also meant that, for the libraries targeted for tuning, it has already been long established as the recommended and most tested compiler. In addition, it is the compiler used for the operating system and system libraries, which makes it especially suitable.

Other compilers were considered where licenses were available, but did not receive so much attention as their use is not so widespread, with several available on only a subset of machine architectures. For example, if looking at the Intel compilers, support for ARM is lacking.

[1] They weren't so well prepared for the first Xeon Phi processors

The Cray compilers, while available towards the end of this project, require a different approach to tuning to that taken in this work. Their key design consideration is to make the life of the programmer easier, and so a lot of work is done that would otherwise be done by the developer within the software's build system. For example, software may be statically analysed and the correct libraries linked in automatically for a range of scientific libraries as well as MPI. The compilers also statically link by default, in contrast with GCC and most other widely used compilers, and may cross-compile where the machine is heterogeneous, which is not an approach taken on the lower-end clusters used at the majority of academic institutions.

A.2 Hardware

Several different machines were used during this project, in particular during the early stages. Due to the period of time over which this work was conducted, some machines that were available at the beginning were taken out of service and replaced before the work was finished.

Although `Balena` A.2.6 was used to produce the results presented in chapter 9, the other machines listed here were all used during the development of one or more of the prototypes.

A.2.1 Aquila

Bath's first production cluster, based on SuperMicro hardware provided by ClusterVision. The installed operating system was Scientific Linux 5.

- 100 nodes;
- 2 sockets per node;
- Intel ®E5462 (Harpertown/Penryn); 4 cores per socket at 2.8GHz
- 16GB RAM per node
- NFS file system;
- DDR InfiniBand 3D torus

Installed in 2006, this system was retired in 2015 and replaced by `Balena`.

A.2.2 Iridis 3

An IBM iDataplex system at the University of Southampton. The operating system installed was RHEL 5.

- More than 11760 Intel ®Nehalem and (later) Westmere processor-cores, providing over 105 TFLOPS R_{peak};
- Slightly more than 1000 nodes, with a total of 22.4 TB RAM;
- DDR & QDR (Mellanox ConnectX) InfiniBand fat tree network;
- IBM General Parallel File System (GPFS), providing approximately 240 TB of usable storage;
- Two 6-core 2.4 GHz Intel ®E5645 (Westmere) processors per node;

- Approximately 22 GB of usable memory per node (the remaining memory is used to store the OS as the nodes are stateless);
- 32 high-memory compute nodes, each with two 4-core 2.27 GHz Intel Nehalem processors and 45 GB of memory available to the users;
- 15 GPU nodes, each with two NVidia Tesla 20-series, M2050, GPU processors (15 TFlops);

This system was installed in 2010 and (mostly) retired in 2016. Access was lost fully in January 2017 when the author changed employer.

A.2.3 Iridis 4

A later IBM iDataplex system, again at the University of Southampton and successor to `Iridis 3`, it was installed with RHEL 6.

- 12320 Cores (250 TFLOPS);
- Two 8-core 2.6 GHz Intel SandyBridge processors per node;
- 4 GB of memory per core;
- 4 high-memory nodes with 32 cores and 256 GB of RAM;
- 24 Intel ®Xeon Phi (5110P) Accelerators (25 TfLOPS) across 12 nodes;
- 1.04 PB of usable storage (GPFS);
- FDR & EDR fat tree (Mellanox) InfiniBand network;

Access to this was lost in January 2017.

A.2.4 Iridis ARM (AKA Amber)

A small cluster comprised of Applied Micro ARMv8a development-board based systems, custom built by/for E4 Systems, who called it the ARKA Server RK004. It was installed in 2014, and consists of:

- 4 nodes (one acting as head node);
- APM X-Gene 1;
- Single socket per node;
- 256GB SSD per node;
- QDR InfiniBand;
- NVidia M40 GPU (though unusable on ARMv8 due to a lack of drivers);

Initially, it was installed with Ubuntu, but later RHEL 7 when `aarch64` support became available. There was a steep learning curve for staff at Southampton, both in using and maintaining it. The differences between this and a now standard `x86_64` machine were greater than expected and often took people by surprise. Few researchers were interested enough to make the high investment of time required to port their software.

Access to this was lost in January 2017.

A.2.5 Iridis PowerPC (AKA black01/black02)

Two IBM POWER 8 machines, though not identical, which were bought for the research community of Southampton to use for porting software and comparisons with other machines. One was equipped with two GPUs, the other with a Nallatech FPGA and supporting software. They are installed with Ubuntu 15.04.

Access to this was also lost in January 2017.

A.2.6 Balena

Another commodity cluster, based on Dell hardware and provided by ClusterVision to the University of Bath in 2015.

- 196 dual-socket (2x 8 cores) Intel®IvyBridge E5-2650 v2 2.60 GHz nodes;
- 17 dual-socket (2x 12 cores) Intel®Skylake (Xeon Gold 6126) 2.60 GHz nodes;
- A small number of nodes with Intel Xeon Phi and GPU (both AMD and NVidia) cards;
- A mixture of 64GB, 128GB or 512GB RAM per node (IvyBridge) or 192GB RAM per node (Skylake);
- Intel Truescale QDR Infiniband configured as a fat tree;
- 630TB BeeGFS storage;

A.2.7 Isambard (Phase 1)

Isambard [164] is a more interesting creation. The result of a collaboration between several universities in the UK and Cray Inc and the UK Met Office. The much anticipated ARMv8 component of this had not yet been delivered when this thesis was completed.

- Network: EDR Infiniband/100GbE
- Nvidia Pascal GPU Nodes (4 nodes)
  - 2 x Intel Xeon E5-2695 v4 2.1 GHz 18 core Broadwell
  - 8 x 16 GB DDR4-2400 MHz
- KNL Nodes (8 nodes)
  - 1 x Intel KNL 7210 1.3 GHz 64 core processor
  - 6 x 16 GB DDR4-2400 MHz

APPENDIX B

THE BEST FLAGS FOUND BY OPTSEARCH

The best flags found by OptSearch v0.9.5 searching over a subset of flags of GCC 7.3.0, targeting the reference BLAS on the Intel®IvyBridge nodes of Balena. This search included some flags that may reduce floating point precision, although the LAPACK BLAS tests, used during the search, passed and HPL reported low error. The precision loss may not be appropriate in all situations, and it is necessary for the user to exercise caution in choice of flags when setting up OptSearch.

Flag:	Included in:
-funroll-all-loops	
-fno-unroll-loops	
-fauto-inc-dec	-O0 -O1 -O2 -O3 -Ofast
-fbranch-count-reg	-O1 -O2 -O3 -Ofast
-fno-branch-probabilities	
-fbranch-target-load-optimize	
-fno-branch-target-load-optimize2	
-fbtr-bb-exclusive	
-fno-caller-saves	
-fno-compare-elim	
-fno-conserve-stack	
-fcprop-registers	-O1 -O2 -O3 -Ofast
-fno-cse-follow-jumps	
-fcx-fortran-rules	
-fcx-limited-range	-Ofast
-fdce	
-fno-delayed-branch	
-fdelete-dead-exceptions	
-fdelete-null-pointer-checks	-O0 -O1 -O2 -O3 -Ofast
-fdevirtualize	-O2 -O3 -Ofast
-fdevirtualize-speculatively	-O2 -O3 -Ofast
-fdse	
-fno-expensive-optimizations	
-fno-forward-propagate	

Flag:	Included in:
-ffp-contract=off	
-ffunction-cse	-O0 -O1 -O2 -O3 -Ofast
-fno-gcse	
-fgcse-after-reload	-O3 -Ofast
-fgcse-las	
-fno-gcse-lm	
-fno-gcse-sm	
-fgraphite	
-fno-graphite-identity	
-fguess-branch-probability	-O1 -O2 -O3 -Ofast
-fno-if-conversion	
-fno-indirect-inlining	
-fno-inline	
-fno-inline-functions	
-fno-inline-small-functions	
-fipa-bit-cp	-O2 -O3 -Ofast
-fno-ipa-cp	
-fipa-cp-alignment	
-fno-ipa-cp-clone	
-fipa-icf-functions	-O2 -O3 -Ofast
-fipa-profile	-O1 -O2 -O3 -Ofast
-fipa-pta	
-fipa-ra	-O2 -O3 -Ofast
-fipa-reference	-O1 -O2 -O3 -Ofast
-fipa-sra	-O2 -O3 -Ofast
-fira-algorithm=CB	
-fno-ira-hoist-pressure	
-fno-ira-loop-pressure	
-fira-region=mixed	
-fno-ira-share-save-slots	
-fno-isolate-erroneous-paths-attribute	
-fno-isolate-erroneous-paths-dereference	
-fjump-tables	
-fkeep-gc-roots-live	
-flimit-function-alignment	
-fno-live-range-shrinkage	
-floop-parallelize-all	
-fno-lra-remat	
-fno-move-loop-invariants	
-fnon-call-exceptions	
-foptimize-sibling-calls	-O2 -O3 -Ofast
-fpartial-inlining	-O2 -O3 -Ofast
-fno-peel-loops	
-fno-plt	
-fprefetch-loop-arrays	-O0 -O1 -O2 -O3 -Ofast
-fno-reciprocal-math	
-freg-struct-return	-O0 -O1 -O2 -O3 -Ofast

Flag:	Included in:
-fno-rename-registers	
-fno-reorder-blocks	
-freorder-blocks-algorithm=simple	
-freorder-blocks-and-partition	
-fno-reschedule-modulo-scheduled-loops	
-frounding-math	
-fno-sched-group-heuristic	
-fno-sched-interblock	
-fno-sched-last-insn-heuristic	
-fno-sched-pressure	
-fsched-rank-heuristic	-O0 -O1 -O2 -O3 -Ofast
-fsched-spec	-O0 -O1 -O2 -O3 -Ofast
-fno-sched-spec-load	
-fsched-spec-load-dangerous	
-fsched-stalled-insns=7904	
-fsched-stalled-insns-dep=5526	
-fno-sched2-use-superblocks	
-fno-schedule-fusion	
-fno-schedule-insns2	
-fsection-anchors	
-fsel-sched-pipelining	
-fsel-sched-pipelining-outer-loops	
-fno-short-enums	
-fno-shrink-wrap	
-fshrink-wrap-separate	-O0 -O1 -O2 -O3 -Ofast
-fsignaling-nans	
-fno-signed-zeros	
-fsimd-cost-model=cheap	
-fsingle-precision-constant	
-fno-split-loops	
-fno-split-paths	
-fstrict-volatile-bitfields	-O0 -O1 -O2 -O3 -Ofast
-fthread-jumps	-O2 -O3 -Ofast
-fno-tree-bit-ccp	
-ftree-builtin-call-dce	-O1 -O2 -O3 -Ofast
-ftree-ch	-O1 -O2 -O3 -Ofast
-fno-tree-cselim	
-fno-tree-dce	
-fno-tree-dominator-opts	
-fno-tree-dse	
-fno-tree-fre	
-ftree-loop-distribute-patterns	-O3 -Ofast
-fno-tree-loop-distribution	
-fno-tree-loop-if-convert	
-fno-tree-loop-if-convert-stores	
-fno-tree-loop-ivcanon	
-ftree-loop-optimize	-O0 -O1 -O2 -O3 -Ofast

Flag:	Included in:
-fno-tree-lrs	
-ftree-parallelize-loops=7761	
-fno-tree-partial-pre	
-fno-tree-pre	
-ftree-pta	-O1 -O2 -O3 -Ofast
-fno-tree-reassoc	
-fno-tree-slsr	
-ftree-switch-conversion	-O2 -O3 -Ofast
-ftree-ter	-O1 -O2 -O3 -Ofast
-fvar-tracking	
-fvar-tracking-uninit	
-fvariable-expansion-in-unroller	
-fkeep-static-consts	-O0 -O1 -O2 -O3 -Ofast
-fno-lto-odr-type-merging	
-fmerge-debug-strings	-O0 -O1 -O2 -O3 -Ofast
-fdwarf2-cfi-asm	-O0 -O1 -O2 -O3 -Ofast
-feliminate-unused-debug-types	-O0 -O1 -O2 -O3 -Ofast
-fshow-column	-O0 -O1 -O2 -O3 -Ofast
-fno-sync-libcalls	
-ftrapping-math	-O0 -O1 -O2 -O3
-fexcess-precision=fast	
-fno-align-labels	
-faggressive-loop-optimizations	-O0 -O1 -O2 -O3 -Ofast
-falign-functions	
-fstore-merging	-O2 -O3 -Ofast
-fssa-backprop	-O0 -O1 -O2 -O3 -Ofast

Table B.1: *Best flags found by OptSearch (v0.9.5) on* `Balena` *for GCC 7.3.0*

B.1 Second best flags found by `OptSearch`, for comparison

The second best set of flags, giving similar results but with far fewer vector instructions. Many of the flags are almost the same as those above.

Flag:	Included in:
-ftree-vectorize	
-fno-tree-vrp	
-fno-unconstrained-commons	
-fno-unroll-all-loops	
-fno-unroll-loops	
-funwind-tables	-O0 -O1 -O2 -O3 -Ofast
-fauto-inc-dec	-O0 -O1 -O2 -O3 -Ofast
-fbranch-count-reg	-O1 -O2 -O3 -Ofast
-fno-branch-probabilities	
-fno-branch-target-load-optimize	
-fbranch-target-load-optimize2	
-fno-btr-bb-exclusive	

Flag:	Included in:
-fcode-hoisting	-O2 -O3 -Ofast
-fno-combine-stack-adjustments	
-fcprop-registers	-O1 -O2 -O3 -Ofast
-fno-cse-follow-jumps	
-fno-cx-fortran-rules	
-fno-cx-limited-range	
-fno-dce	
-fdelayed-branch	
-fno-delete-dead-exceptions	
-fdelete-null-pointer-checks	-O0 -O1 -O2 -O3 -Ofast
-fno-dse	
-fno-exceptions	
-ffast-math	
-ffinite-math-only	-Ofast
-fno-float-store	
-fforward-propagate	-O1 -O2 -O3 -Ofast
-ffp-contract=off	
-ffp-int-builtin-inexact	-O0 -O1 -O2 -O3 -Ofast
-fno-gcse	
-fgcse-after-reload	-O3 -Ofast
-fno-gcse-lm	
-fgcse-sm	
-fno-graphite	
-fgraphite-identity	
-fno-hoist-adjacent-loads	
-fif-conversion	-O1 -O2 -O3 -Ofast
-fno-inline	
-finline-functions	-O1 -O2 -O3 -Ofast
-fno-ipa-cp-alignment	
-fno-ipa-icf	
-fno-ipa-icf-variables	
-fipa-profile	-O1 -O2 -O3 -Ofast
-fno-ipa-pta	
-fno-ipa-vrp	
-fira-algorithm=priority	
-fira-hoist-pressure	-O0 -O1 -O2 -O3 -Ofast
-fno-ira-loop-pressure	
-fira-region=one	
-fno-ira-share-save-slots	
-fno-ira-share-spill-slots	
-fisolate-erroneous-paths-attribute	
-fno-isolate-erroneous-paths-dereference	
-fno-ivopts	
-fjump-tables	
-fno-keep-gc-roots-live	
-fno-limit-function-alignment	
-fno-loop-nest-optimize	

Flag:	Included in:
-fno-loop-parallelize-all	
-fno-modulo-sched	
-fmodulo-sched-allow-regmoves	
-fno-move-loop-invariants	
-fno-non-call-exceptions	
-fno-opt-info	
-foptimize-sibling-calls	-O2 -O3 -Ofast
-fno-partial-inlining	
-fno-peel-loops	
-fpeephole	-O0 -O1 -O2 -O3 -Ofast
-fpeephole2	-O2 -O3 -Ofast
-fprefetch-loop-arrays	-O0 -O1 -O2 -O3 -Ofast
-fno-reciprocal-math	
-freg-struct-return	-O0 -O1 -O2 -O3 -Ofast
-freorder-blocks	-O1 -O2 -O3 -Ofast
-fno-reorder-functions	
-fno-rerun-cse-after-loop	
-fno-reschedule-modulo-scheduled-loops	
-fno-rounding-math	
-fsched-critical-path-heuristic	-O0 -O1 -O2 -O3 -Ofast
-fsched-group-heuristic	-O0 -O1 -O2 -O3 -Ofast
-fno-sched-interblock	
-fno-sched-spec-load-dangerous	
-fsched-stalled-insns=6756	
-fsched-stalled-insns-dep=3230	
-fno-schedule-fusion	
-fschedule-insns	-O2 -O3 -Ofast
-fschedule-insns2	-O2 -O3 -Ofast
-fno-sel-sched-pipelining-outer-loops	
-fno-shrink-wrap-separate	
-fsigned-zeros	-O0 -O1 -O2 -O3
-fsimd-cost-model=dynamic	
-fsplit-wide-types	-O1 -O2 -O3 -Ofast
-fstrict-volatile-bitfields	-O0 -O1 -O2 -O3 -Ofast
-fno-tracer	
-ftree-ch	-O1 -O2 -O3 -Ofast
-fno-tree-coalesce-vars	
-fno-tree-copy-prop	
-fno-tree-dce	
-fno-tree-dominator-opts	
-ftree-forwprop	-O0 -O1 -O2 -O3 -Ofast
-fno-tree-fre	
-ftree-loop-ivcanon	-O0 -O1 -O2 -O3 -Ofast
-ftree-loop-optimize	-O0 -O1 -O2 -O3 -Ofast
-ftree-loop-vectorize	-O3 -Ofast
-ftree-lrs	
-ftree-parallelize-loops=7141	

Flag:	Included in:
-ftree-partial-pre	-O3 -Ofast
-fno-tree-phiprop	
-fno-tree-pre	
-fno-tree-pta	
-ftree-sink	-O1 -O2 -O3 -Ofast
-ftree-slp-vectorize	-O3 -Ofast
-fno-tree-slsr	
-ftree-switch-conversion	-O2 -O3 -Ofast
-fno-tree-tail-merge	
-ftree-ter	-O1 -O2 -O3 -Ofast
-fvar-tracking-assignments	
-fno-var-tracking-assignments-toggle	
-fno-variable-expansion-in-unroller	
-fvect-cost-model=dynamic	
-fno-keep-static-consts	
-flto-odr-type-merging	-O0 -O1 -O2 -O3 -Ofast
-fmerge-constants	-O1 -O2 -O3 -Ofast
-fno-merge-debug-strings	
-fno-semantic-interposition	
-fno-show-column	
-ftoplevel-reorder	-O1 -O2 -O3 -Ofast
-fno-align-jumps	
-fno-align-labels	
-fno-aggressive-loop-optimizations	
-fno-align-functions	
-fstdarg-opt	-O0 -O1 -O2 -O3 -Ofast
-fstore-merging	-O2 -O3 -Ofast
-fssa-backprop	-O0 -O1 -O2 -O3 -Ofast
-fno-ssa-phiopt	

Table B.2: *Second best flags found by OptSearch (v0.9.3) on `Balena` for GCC 7.3.0*

APPENDIX C

HPL INPUT DATA USED ON BALENA

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out output file name (if any)
6 device out (6=stdout,7=stderr,file)
1 # of problems sizes (N)
9600 Ns # N on one node
1 # of NBs
100 NBs # should really be 256 for this processor
0 PMAP process mapping (0=Row-,1=Column-major)
1 # of process grids (P x Q)
4 Ps
4 Qs
16.0 threshold
1 # of panel fact
2 PFACTs (0=left, 1=Crout, 2=Right)
1 # of recursive stopping criterium
2 NBMINs (>= 1)
1 # of panels in recursion
2 NDIVs
1 # of recursive panel fact.
2 RFACTs (0=left, 1=Crout, 2=Right)
1 # of broadcast
0 BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1 # of lookahead depth
0 DEPTHs (>=0)
2 SWAP (0=bin-exch,1=long,2=mix)
64 swapping threshold
0 L1 in (0=transposed,1=no-transposed) form
0 U in (0=transposed,1=no-transposed) form
1 Equilibration (0=no,1=yes)
8 memory alignment in double (> 0)
##### This line (no. 32) is ignored (it serves as a separator). ######
0 Number of additional problem sizes for PTRANS
1200 10000 30000 values of N
0 number of additional blocking sizes for PTRANS
40 9 8 13 13 20 16 32 64 values of NB
```

APPENDIX D

AN EMAIL FROM ONE OF THE USERS OF AQUILA

What follows is an email from Marco Molinari, a PGR in Chemistry at the University of Bath at that time, since graduated. The author was tasked with looking after the production HPC system, `Aquila`. This job included the building of scientific software applications and libraries for those who required them. (The full software stack took more than 6 months to build, during which time several new versions of applications and their supporting libraries would have appeared, and the cycle would start all over again.)

Note that for brevity, the conversation history has been omitted. Molinari had requested assistance in compiling DL_POLY on `Aquila`, and after a few difficulties in understanding what needed to be done, the author sent him a copy of the Makefile, edited already, and the instruction to run `make hpc`. This is his email confirming that this new target had worked better than expected.

The real difference was not supporting libraries he had chosen, or even the compiler version. It was which flags were passed to the compiler, instructing it to perform optimizations suitable for this particular application that it would not have done otherwise.

```
Date: Thu, 10 Nov 2011 13:23:21 +0000
From: "marco.molinari" <m.molinari@bath.ac.uk>
To: J Jones <j.jones@bath.ac.uk>
Subject: Re: DLPOLY

Hi Jess,

I compiled dlpoly 2 with your makefile! It is much faster than all the
compilation I have tried on Aquila! Thank you.

Marco

--
*******************************************
Marco Molinari
Research Officer
Department of Chemistry
```

University of Bath
Claverton Down
Bath BA2 7AY
United Kingdom

Email: m.molinari@bath.ac.uk
Telephone: 0044 (0)1225 386523
****

APPENDIX E

AN EXAMPLE CONFIGURATION FILE FOR OPTSEARCH

```
---
## Main config file for optsearch
# These are simple scripts intended to make configuration slightly easier.
quit-signal: SIGUSR1 # see signal(7) manpage for the list of signals. By default slurm will send SIGTERM 3
clean-script: ./balena/clean-blas.sh # Clean the build directory, etc, before beginning each run
build-script: ./balena/build-blas.sh # equiv to make; must use environment variable FLAGS, set by OptSearc
accuracy-test: ./balena/test-blas.sh # Run some tests to check that numerical results are not adversely af
performance-test: ./balena/run-hpl.sh # Run a benchmark. Return a measurement, such as run time, that can
timeout: 360 # How long to wait for commands to run before killing the spawned process, in seconds
epsilon: 3.0 # Allowed/expected experimental error (% of mean runtime)
benchmark-timeout: 3600 # A separate timeout for the benchmark
benchmark-repeats: 20 # Maximum number of times to repeat the benchmark
# Compiler specific settings:
compiler:
 name: gfortran
 version: 7.3.0
 flags:
 - name: predictable-branch-outcome
 type: range
 max: 50
 min: 0
 prefix: '--param '
 separator: '='
 - name: partial-inlining-entry-probability
 type: range
 max: 100
 min: 0
 prefix: '--param '
 separator: '='
 - name: hot-bb-count-ws-permille
 type: range
 max: 1000
 min: 0
 prefix: '--param '
 separator: '='
 - name: unlikely-bb-count-fraction
 type: range
 max: 10000
 min: 1
 prefix: '--param '
 separator: '='
 - name: stack-reuse
 type: list
 prefix: '-f'
 separator: '='
 values: [ 'all', 'named_vars', 'none' ]
```

```
- name: tree-vectorize
type: on-off
on-prefix: '-f'
off-prefix: '-fno-'
- name: tree-vrp
type: on-off
on-prefix: '-f'
off-prefix: '-fno-'
- name: code-hoisting
type: on-off
on-prefix: '-f'
off-prefix: '-fno-'
- name: combine-stack-adjustments
type: on-off
on-prefix: '-f'
off-prefix: '-fno-'
- name: compare-elim
type: on-off
on-prefix: '-f'
off-prefix: '-fno-'
- name: conserve-stack
type: on-off
on-prefix: '-f'
off-prefix: '-fno-'
- name: cprop-registers
type: on-off
on-prefix: '-f'
off-prefix: '-fno-'
- name: crossjumping
type: on-off
on-prefix: '-f'
off-prefix: '-fno-'
- name: cse-follow-jumps
type: on-off
on-prefix: '-f'
off-prefix: '-fno-'
- name: cx-fortran-rules
type: on-off
on-prefix: '-f'
- name: sched-stalled-insns
type: range
prefix: '-f'
separator: '='
min: 0
max: 8192
- name: sched-stalled-insns-dep
type: range
prefix: '-f'
separator: '='
min: 0
max: 8192
off-prefix: '-fno-'
```

APPENDIX F

DETAILED WORKINGS FOR ESTIMATE OF ECONOMIC VALUE

In [17], the authors investigate the economic value of hand-optimizing, stating:

> "In our study, we pessimistically assume improvements from 5% upwards, and even at the low end, an investment in "brainware" for HPC pays off."

In chapter 9, we demonstrated a performance improvement of 24% over `-O3`, which is almost 5 times more.

Bischof et al additionally assume 2 months of work is required for a performance expert to optimize each of their codes, and that these user applications will benefit for 2 years. This makes their savings estimate pessimistic if the performance tuning work can be undertaken when the machine is commissioned, since most HPCs at academic institutions are used running production workloads for 5 years or more (as they admit in [17, page 2]). However, given the usual time to availability of high performance BLAS libraries, 2 years seems fitting if only that library is being considered. Once libraries delivering higher performance are available they should be used instead.

They give the cost for tuning the top 18 projects, responsible for 50% of core hours, as being equivalent to 1.5 years of one FTE. For an auto-tuner, this cost should be substantially lower: The preparations described in sections 8.2 and 8.3 took approximately one working day for one person familiar with `OptSearch`. The tuning itself took around 1-2 hours using 4 nodes of the system, which has little impact on production job throughput, and indicates that a keen system administrator could potentially tune several libraries simultaneously when the machine is newly installed and thus largely unoccupied by production jobs.

In [7], 50% of core hours are used by 20 applications. Of these top 20 applications, 11 use dense linear algebra, and so would likely benefit from an optimized BLAS library. Using this to provide a ratio for the number of applications likely to be benefiting from auto-tuning suggests (if all applications within that 20 are equally heavily used) that 27.5% of core hours are used by such applications.

$0.24 \times 0.275 \times \text{TCO of the machine} = 0.066 \times \text{TCO}$

Using the total cost of ownership (TCO) figure for RWTH Aachen calculated by Bischof et al, this comes to around 363,000€ (just over $471,000, using the December 2012 exchange rate) *for tuning only the BLAS library*. However, this simplistic calculation assumes that the auto-tuning has no cost. It therefore may be better considered in terms of CPU-hours.

The June 2012 TOP500 gives the details of the RWTH Aachen machine (78 in the list at that time), a Bullx B500 cluster, thus:

Site:	Universitaet Aachen/RWTH
Cores:	25,448
Processor:	Intel ®Xeon X5675 6C 3.06GHz
Interconnect:	Infiniband QDR
HPL Performance (R_{max}):	219.838 TFLOP/s
Theoretical Peak (R_{peak}):	270.538 TFLOP/s
N_{max}:	2,145,920

Bischof et al give a very optimistic 24/7 availability for their machine, and do not give any indication of how many nodes may be out of service, nor do they state the frequency or duration of planned maintenance windows. It seems unrealistic to assume 100% availability during a year.

Many supercomputing centres aim for 80% utilisation, and plan for a small amount of planned down-time for maintenance. If it is assumed that 80% of the core hours are used for 360 days of the year (probably a slightly optimistic estimate, but possible), we can adjust the calculation accordingly.

% performance improvement $\times$ % of top applications using BLAS $\times$ available CPU-hours
$= 0.24 \times 0.275 \times$ available CPU-hours
$= 0.066 \times (25448 \times 0.8 \times 360 \times 24)$
$= 11609174$ CPU-hours

We can now subtract the tiny cost of auto-tuning on 4 nodes for 1.5 hours, to give a representative figure for the number of CPU hours returned to the users of the system.

$= 11609174 - (4 \times 12 \times 1.5)$
$= 11609174 - 72$
$= 11609102$ CPU-hours

GLOSSARY

AI Artificial Intelligence. 24

API Application Programming Interface. 36, 93, 117

baselining The empirical discovery of the baseline performance of a benchmark on a particular supercomputer, used during performance tuning and commercial performance evaluation prior to optimization work. 62

BLAS Basic Linear Algebra Subprograms. 18, 35–39, 43, 48–52, 65, 96, 100, 104, 108, 112, 115–117, 120, 124, 136

DGEMM Double-precision general matrix multiplication algorithm, part of the level 3 BLAS. 112

DMA direct memory access. 24

DTLB data translation look-aside buffer. 51

FFT fast Fourier transform. 38, 117

FLOPS floating-point operations per second. 24, 26

FMA Fused Multiply-Add. 22, 24

FPU Floating Point Unit. 26, 31

GCC GNU Compiler Collection. 10, 17, 34, 35, 44, 54–56, 60, 65, 75, 86–88, 92, 100, 101, 103, 104, 108, 109, 111, 113, 115, 117, 124, 127, 130

GEMM general matrix multiplication algorithm. 112

GPGPU General Purpose Graphics Processing Unit. 23

GPU Graphics Processing Unit. 22–24, 31

HPC High Performance Computer. 6, 14–17, 19, 22, 24, 27, 34, 41–43, 45, 48, 52–56, 58, 59, 75, 77, 89, 90, 110, 112, 116, 117, 136

HPCG High Performance Conjugate Gradients. 116

HPL High Performance LINPACK. 7, 13, 24, 36, 37, 39, 43, 48–51, 65, 66, 72, 96, 100–104, 108, 112, 116, 124, 131, 137

IB InfiniBand. 25

LAPACK Linear Algebra PACKage. 36, 96, 100, 124

LLC last level cache. 50

LLVM Low Level Virtual Machine. 34, 35, 44, 111

MAC Multiply-Accumulate. 22

MKL Math Kernel Library. 20, 47, 50

ML Machine Learning. 24, 42

MPI Message Passing Interface. 25, 39, 57, 59, 75, 77–79, 84, 86, 92, 96–99, 101, 102, 116

NUMA Non-Uniform Memory Access. 31

OOM Out Of Memory. 89

PRNG pseudo-random number generator. 5, 78, 84, 91, 93

PSO Particle Swarm Optimization. 6, 13, 76, 80, 81, 84, 92, 101, 104, 106, 109, 113

RSE Research Software Engineer. 15

SBLAS Sparse Basic Linear Algebra Subprograms. 112

SIMD Single Instruction, Multiple Data. 22, 23, 26, 35, 38, 88, 102, 104

SMP Symmetric Multi-Processor. 25, 30

SMT Simultaneous Multi-Threading. 31

SPSO Standard Particle Swarm Optimization. 58, 77, 78, 80, 81, 90, 91, 101, 105, 106, 109, 113

SSI Software Sustainability Institute. 15

TDP thermal design power. 27

TLB translation look-aside buffer. 5, 28–30, 50, 51

WAL Write-Ahead Logging. 93

BIBLIOGRAPHY

[1] External clang examples: A list of projects and tools, 2018. URL https://clang.llvm.org/docs/ExternalClangExamples.html.

[2] Software sustainability institute (SSI). https://www.software.ac.uk/, 2018. URL https://www.software.ac.uk/.

[3] Lefohn Aaron, Joe Kniss, and John Owens. GPU gems - chapter 33. implementing efficient parallel data structures on GPUs. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter33.html, April 2005. URL http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter33.html.

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[5] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong. Program energy efficiency: The impact of language, compiler and implementation choices. In *International Green Computing Conference*, pages 1–6, Nov 2014. ISBN 978-1-4799-6177-1. doi: 10.1109/IGCC.2014.7039169.

[6] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 231–239, New York, NY, USA, 2004. ACM. ISBN 1-58113-806-7. doi: 10.1145/997163.997196. URL http://doi.acm.org/10.1145/997163.997196.

[7] V Anantharaj, F Foertter, W Joubert, and J Wells. Approaching exascale: Application requirements for OLCF leadership computing. Technical report, Oak Ridge National Lab, 2013. URL https://www.olcf.ornl.gov/wp-content/uploads/2013/01/OLCF_Requirements_TM_2013_Final1.pdf. 11 of their top 20 applications use dense LA.

[8] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 303–316, Edmonton, Canada, August 2014. doi: 10.1145/2628071.2628092.

[9] Arm Ltd. ARM ® architecture reference manual supplement: The scalable vector extension (SVE), for ARMv8-A. https://static.docs.arm.com/ddi0584/a/DDI0584A_b_SVE_supp_armv8A.pdf, 2017. URL https://static.docs.arm.com/ddi0584/a/DDI0584A_b_SVE_supp_armv8A.pdf. page 5-49 describes complex SIMD instructions.

[10] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[11] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5):96:1–96:42, September 2018. ISSN 0360-0300. doi: 10.1145/3197978. URL http://doi.acm.org/10.1145/3197978.

[12] David H. Bailey, Samuel Williams, Kaushik Datta, Jonathon Carter, Leonid Oliker, John Shalf, and Katherine A. Yelick. PERI - auto-tuning memory intensive kernels for multicore. Seattle, WA. US, June 2008. URL http://www.osti.gov/bridge/product.biblio.jsp?osti_id=936521.

[13] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, PP(99):1–16, July 2018. ISSN 0018-9219. doi: 10.1109/JPROC.2018.2841200. URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8423171.

[14] Shajulin Benedict. Prediction assisted runtime based energy tuning mechanism for HPC applications. *Sustainable Computing: Informatics and Systems*, 2018. ISSN 2210-5379. doi: 10.1016/j.suscom.2018.06.004. URL http://www.sciencedirect.com/science/article/pii/S2210537916302335.

[15] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1):43–56, 1995. ISSN 0010-4655. doi: 10.1016/0010-4655(95)00042-E. URL http://www.sciencedirect.com/science/article/pii/001046559500042E.

[16] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th international conference on Supercomputing*, pages 340–347. ACM, 1997.

[17] Christian Bischof, Dieter An Mey, and Christian Iwainsky. Brainware for green HPC. *Computer Science - Research and Development*, 27(4):227–233, December 2012. ISSN 1865-2034. doi: 10.1007/s00450-011-0198-5. URL `http://dx.doi.org/10.1007/s00450-011-0198-5`.

[18] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. ISBN 0-89871-397-8 (paperback).

[19] James Bornholt. How not to measure supercomputing performance. `https://homes.cs.washington.edu/~bornholt/post/performance-evaluation.html`, November 2014. URL `https://homes.cs.washington.edu/~bornholt/post/performance-evaluation.html`.

[20] George Bosilca, Zizhong Chen, Jack Dongarra, Victor Eijkhout, Graham E Fagg, Erika Fuentes, Julien Langou, Piotr Luszczek, Jelena Pjesivac-Grbovic, Keith Seymour, Haihang You, and Sathish S. Vadhiyar. Self-adapting numerical software (sans) effort. *IBM Journal of Research and Development*, 50(2.3):223–238, 2006.

[21] Mohamed Boussaa. *Automatic Non-functional Testing and Tuning of Configurable Generators*. Theses, Inria Rennes - Bretagne Atlantique ; University of Rennes 1, September 2017. URL `https://hal.archives-ouvertes.fr/tel-01598821`.

[22] various contributors Brendan Gregg. perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`, 2015. URL `https://perf.wiki.kernel.org/index.php/Main_Page`.

[23] Chris Budd. Software review and hardware refresh cycles of the uk met office. Private communication to JHD, 2017.

[24] J Mark Bull. Single node performance analysis of applications on HPCx. Technical HPCxTR0703, STFC, Daresbury, 2007. URL `http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0703.pdf`.

[25] C11 Standard. ISO/IEC 9899:2011 programming languages – C. `https://www.iso.org/standard/57853.html`, 2011. URL `https://www.iso.org/standard/57853.html`.

[26] C99 Standard. ISO/IEC 9899:1999 programming languages – C. `https://www.iso.org/standard/29237.html`, 1999. URL `https://www.iso.org/standard/29237.html`.

[27] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 161–168, New York, NY, USA, 2006. ACM. ISBN 1-59593-383-2. doi: 10.1145/1143844.1143865. URL `http://doi.acm.org/10.1145/1143844.1143865`.

[28] T.C.E. Cheng and Mikhail Y. Kovalyov. An unconstrained optimization problem is NP-hard given an oracle representation of its objective function: a technical note. *Computers & Operations Research*, 29(14):2087–2091, 2002. ISSN 0305-0548. doi: https://doi.org/10.1016/S0305-0548(02)00065-5. URL http://www.sciencedirect.com/science/article/pii/S0305054802000655.

[29] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. Run-to-run variability on xeon phi based cray xc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 52:1–52:13, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5114-0. doi: 10.1145/3126908.3126926. URL http://doi.acm.org/10.1145/3126908.3126926.

[30] Maurice Clerc. Confinements and Biases in Particle Swarm Optimisation. 9 pages, 2006. URL https://hal.archives-ouvertes.fr/hal-00122799.

[31] Maurice Clerc. Stagnation Analysis in Particle Swarm Optimisation or What Happens When Nothing Happens. 17 pages, December 2006. URL https://hal.archives-ouvertes.fr/hal-00122031.

[32] Maurice Clerc. Randomness matters. Technical report, independent researcher, May 2012. URL https://hal.archives-ouvertes.fr/hal-00764990. 14 pages.

[33] Maurice Clerc. Standard particle swarm optimisation. 15 pages, September 2012. URL https://hal.archives-ouvertes.fr/hal-00764996.

[34] Phil Colella. Defining software requirements for scientific computing. http://view.eecs.berkeley.edu/w/images/temp/6/6e/20061003235551!DARPAHPCS.ppt, 2004.

[35] Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Exploring the structure of the space of compilation sequences using randomized search algorithms. *The Journal of Supercomputing*, 36(2):135–151, 2006. ISSN 1573-0484. doi: 10.1007/s11227-006-7954-5. URL https://doi.org/10.1007/s11227-006-7954-5.

[36] Simon J Cox, James T Cox, Richard P Boardman, Steven J Johnston, Mark Scott, and Neil S O'Brien. Iridis-Pi: A low-cost, compact demonstration cluster. https://www.southampton.ac.uk/~sjc/raspberrypi/raspberry_pi_iridis_lego_supercomputer_paper_cox_Jun2013.pdf, February 2013. URL https://www.southampton.ac.uk/~sjc/raspberrypi/raspberry_pi_iridis_lego_supercomputer_paper_cox_Jun2013.pdf.

[37] Cray Compiler. Released: Cray XC programming environments 18.08. https://pubs.cray.com/content/00628300-DB/DB00628299, August 2018. URL https://pubs.cray.com/content/00628300-DB/DB00628299.

[38] Cray "Old Hands". Early memories of Cray Research. Private communication to JRJ, 2018. Several of the "old hands" at Chippewa originally worked for CDC and/or UniSYS, and remained with the company when it was bought by SGI and later spun

out again. They were kind enough to answer questions about old hardware. One, who worked for Burroughs as a young man, was able to explain their hardware in detail.

[39] P. De, R. Kothari, and V. Mann. Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *2007 IEEE International Conference on Cluster Computing*, pages 331–340, Sept 2007. doi: 10.1109/CLUSTR.2007.4629247.

[40] Pablo de Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. CERE: LLVM Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12 (1):6, 2015. doi: 10.1145/2724717.

[41] Docker Containers. Docker containers. `https://www.docker.com/`, 2012. URL `https://www.docker.com/`.

[42] J Dongarra, J Bunch, C Moler, and G W Stewart. *LINPACK Users' Guide*. Philadelphia, Pennsylvania, United States, 1979.

[43] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, March 1990. ISSN 00983500. doi: 10.1145/77626.79170. URL `http://dl.acm.org/citation.cfm?id=79170`.

[44] Jack Dongarra. The HPC challenge benchmark: A candidate for replacing LINPACK in the top500? `https://www.spec.org/workshops/2007/austin/slides/Keynote_Jack_Dongarra.pdf`, January 2007. URL `https://www.spec.org/workshops/2007/austin/slides/Keynote_Jack_Dongarra.pdf`. Slide 9 mentions problems with HPL, though NOT compiler writers gaming the system yet.

[45] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. The high performance conjugate gradients benchmark. `http://hpcg-benchmark.org/`, 2014. URL `http://hpcg-benchmark.org/`.

[46] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. A new metric for ranking high-performance computing systems. *National Science Review*, 3(1):30–35, 2016. URL `https://academic.oup.com/nsr/article-abstract/3/1/30/2460324`.

[47] J.J. Dongarra. High performance computing today and benchmarking the future. `http://www.hpcc.unical.it/hpc2014/pdfs/dongarra.pdf`, 2014.

[48] I. Duff, M. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM TOMS*, 28(2):239–267, June 2002.

[49] Russ C Eberhart, James Kennedy, et al. A new optimizer using particle swarm theory. In *Proceedings of the sixth international symposium on micro machine and human science*, volume 1, pages 39–43. New York, NY, 1995.

[50] Michael Ellerman. [GIT PULL] please pull powerpc/linux.git powerpc-4.6-1 tag. `http://lkml.iu.edu/hypermail/linux/kernel/1603.2/02109.`

html, March 2016. URL http://lkml.iu.edu/hypermail/linux/kernel/1603.2/02109.html. Patches submitted to Linux kernel for POWER9 architecture support, a year before the chips became available to buy; they were announced by IBM in August 2016.

[51] EPCC. ARCHER: Scientific software packages. http://www.archer.ac.uk/documentation/software/, 2018. URL http://www.archer.ac.uk/documentation/software/.

[52] JJ Erpenbeck, WW Wood, and BJ Berne. Statistical mechanics. part b: Time-dependent processes. In *Modern Theoretical Chemistry*, volume 6. Plenum New York, 1977.

[53] Massimiliano Fatica. Accelerating linpack with CUDA on heterogenous clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 46–51, Washington, D.C., 2009. ACM. ISBN 978-1-60558-517-8. doi: 10.1145/1513895.1513901. URL http://portal.acm.org/citation.cfm?id=1513901.

[54] Fedora Linux. The fedora linux project. https://getfedora.org/, 2017. URL https://getfedora.org/.

[55] Thomas S. Ferguson. A bayesian analysis of some nonparametric problems. *The Annals of Statistics*, 1(2):209–230, 03 1973. doi: 10.1214/aos/1176342360. URL https://doi.org/10.1214/aos/1176342360.

[56] Agner Fogg. Test results for broadwell and skylake. http://agner.org/optimize/blog/read.php?i=415, 2015. URL http://agner.org/optimize/blog/read.php?i=415.

[57] Agner Fogg. What is the status of VZEROUPPER use? https://software.intel.com/en-us/forums/intel-isa-extensions/topic/704023, 2016. URL https://software.intel.com/en-us/forums/intel-isa-extensions/topic/704023.

[58] Agner Fogg. Agner's CPU blog – test results for Knights Landing. https://www.agner.org/optimize/blog/read.php?i=761, November 2016. URL https://www.agner.org/optimize/blog/read.php?i=761.

[59] Fortran 2010 Standard. ISO/IEC 1539-1:2010 information technology – programming languages – Fortran – part 1: Base language. https://www.iso.org/standard/50459.html, 2010. URL https://www.iso.org/standard/50459.html.

[60] The MPI Forum. Message passing interface forum. http://www.mpi-forum.org/, March 2009. URL http://www.mpi-forum.org/.

[61] The Raspberry Pi Foundation. Raspberry pi. https://www.raspberrypi.org/, May 2009. URL https://www.raspberrypi.org/.

[62] M. Frigo and S. G Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, 1998*, volume 3, pages 1381–1384 vol.3. IEEE, May 1998. ISBN 0-7803-4428-6. doi: 10.1109/ICASSP.1998.681704.

[63] John L. Furlani and Peter W. Osel. Abstract yourself with modules. In *Proceedings of the Tenth Large Installation Systems Administration Conference (LISA '96)*, number 10, pages 193–204, 1996. URL `http://modules.sourceforge.net/docs/absmod.pdf`.

[64] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*, Ottawa, Canada, June 2008. URL `https://hal.inria.fr/inria-00294704`.

[65] V. Gazi. Asynchronous particle swarm optimization. In *2007 IEEE 15th Signal Processing and Communications Applications*, pages 1–4, June 2007. doi: 10.1109/SIU.2007.4298806.

[66] GNU. GNU binutils. `https://www.gnu.org/software/binutils/`, September 1990. URL `https://www.gnu.org/software/binutils/`.

[67] Kazushige Goto and Robert van de Geijn. On reducing TLB misses in matrix multiplication. 2002. doi: 10.1.1.12.4905. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.4905`.

[68] Kazushige Goto and Robert van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3), May 2008. ISSN 0098-3500. doi: 10.1145/1356052.1356053.

[69] Kazushige Goto and Robert van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)*, 35(1), July 2008. ISSN 0098-3500. doi: 10.1145/1377603.1377607. URL `http://portal.acm.org/citation.cfm?id=1377607`.

[70] Graph500 List. The graph500 list. `https://graph500.org/`, November 2010. URL `https://graph500.org/`.

[71] The MVAPICH Group. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. `http://mvapich.cse.ohio-state.edu/`, 2001. URL `http://mvapich.cse.ohio-state.edu/`.

[72] 754 WG Working group for floating-point arithmetic. 754 – standard for floating-point arithmetic. `https://standards.ieee.org/develop/project/754.html`, 1985. URL `https://standards.ieee.org/develop/project/754.html`.

[73] Per Brinch Hansen. *Classic Operating Systems*. Springer Verlag, 2000. ISBN 0-387-95113-X.

[74] Mark Harris, Michael Garland, Nadathur Satish, and Shubho Sengupta. thrust - google code. `http://code.google.com/p/thrust/`, July 2009. URL `http://code.google.com/p/thrust/`.

[75] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*, pages 1 –11, May 2009. doi: 10.1109/IPDPS.2009.5161004.

[76] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 84:1–84:11, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-4673-8815-3. URL http://dl.acm.org/citation.cfm?id=3014904.3015017.

[77] Sabine Helwig and Rolf Wanka. Particle swarm optimization in high-dimensional bounded search spaces. In *2007 IEEE Swarm Intelligence Symposium*, pages 198–205. IEEE, 2007.

[78] John L Hennessey and David A Patterson. *Computer architecture: A quantitative approach*. Elsevier Inc, 1990.

[79] Greg Henry. BLAS Based on Block Data Structures. Technical Report 89, Cornell University, January 1992. http://ecommons.library.cornell.edu/bitstream/1813/5471/1/92-089.pdf.

[80] Simon Hettrick. It's impossible to conduct research without software, say 7 out of 10 UK researchers. https://www.software.ac.uk/blog/2014-12-04-its-impossible-conduct-research-without-software-say-7-out-10-uk-researchers, December 2014. URL https://www.software.ac.uk/blog/2014-12-04-its-impossible-conduct-research-without-software-say-7-out-10-uk-researchers.

[81] Mark D Hill. Aspects of cache memory and instruction buffer performance. Technical report, California University, Berkeley. Department of Electrical Engineering and Computer Sciences, 1987.

[82] D. Richard Hipp, Dan Kennedy, and Joe Mistachkin. SQLite database engine. https://sqlite.org/index.html, May 2000. URL https://sqlite.org/index.html.

[83] D. Richard Hipp, Dan Kennedy, and Joe Mistachkin. Write-ahead logging, 07 2010. URL https://sqlite.org/wal.html.

[84] Kenneth Hoste and Lieven Eeckhout. COLE: Compiler optimization level exploration. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 165–174, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4. doi: 10.1145/1356058.1356080. URL http://doi.acm.org/10.1145/1356058.1356080.

[85] IBM Compilers. IBM compilers - overview. https://www.ibm.com/us-en/marketplace/ibm-compilers, 2018. URL https://www.ibm.com/us-en/marketplace/ibm-compilers.

[86] Cadence Design Systems Inc. Tensilica customizable processors. https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable, 2019. URL https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable.

[87] Graphcore Inc. Graphcore: Accelerating machine learning. https://www.graphcore.ai/, 2018. URL https://www.graphcore.ai/.

[88] Intel. Math kernel library from intel, May 2003. URL http://software.intel.com/en-us/articles/intel-mkl/.

[89] Intel. Intel Xeon Processor E5 V2 Family Specification Update. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v2-spec-update.pdf, September 2015. URL https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v2-spec-update.pdf.

[90] Intel. Intel Xeon Processor E7 v2 Family Specification Update. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e7-v2-spec-update.pdf, September 2017. URL https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e7-v2-spec-update.pdf.

[91] Intel. Intel Xeon Processor E5 v3 Family Specification Update. https://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-v3-spec-update.html, September 2017. URL https://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-v3-spec-update.html.

[92] Intel. Intel Xeon Processor Scalable Family Specification Update. https://www.intel.co.uk/content/www/uk/en/processors/xeon/scalable/xeon-scalable-spec-update.html, February 2018. URL https://www.intel.co.uk/content/www/uk/en/processors/xeon/scalable/xeon-scalable-spec-update.html.

[93] Intel. Intel Xeon Processor E3 v5 Family Specification Update. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v5-spec-update.pdf, August 2018. URL https://www.intel.co.uk/content/www/uk/en/processors/xeon/xeon-e3-1200v3-spec-update.html.

[94] Martyn Corden (Intel). Consistency of floating-point results using the Intel compiler. https://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler/, August 2012. URL https://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler/.

[95] Martyn Corden (Intel). Differences in floating-point arithmetic between Intel Xeon processors and the Intel Xeon Phi Coprocessor x100 product family. https://software.intel.com/en-us/articles/differences-in-floating-point-arithmetic-between-intel-xeon-processors-and-the-intel-xeon, March 2013. URL https://software.intel.com/en-us/articles/differences-in-floating-point-

arithmetic-between-intel-xeon-processors-and-the-intel-xeon.

[96] Intel®. Intel®optimization notice. https://software.intel.com/en-us/articles/optimization-notice/, August 2012. URL https://software.intel.com/en-us/articles/optimization-notice/.

[97] French Alternative Energies Jean-Philippe Nominé and Atomic Energy Commission (CEA). Exascale leaders on next horizons in supercomputing. https://www.nextplatform.com/2017/02/23/exascale-leaders-look-next-horizons-supercomputing/, February 2017. URL https://www.nextplatform.com/2017/02/23/exascale-leaders-look-next-horizons-supercomputing/.

[98] J.R. Jones, J.H. Davenport, and R. Bradford. The changing relevance of the tlb. In *Distributed Computing and Applications to Business, Engineering Science (DCABES), 2013 12th International Symposium on*, pages 110–114, Sept 2013. doi: 10.1109/DCABES.2013.27.

[99] Frederick P Brooks Jr. *The Mythical Man-Month: Essays on software engineering*. Addison-Wesley, 1975. Reprinted with corrections, January 1982.

[100] William Kahan. The baleful effect of computer languages and benchmarks upon applied mathematics, physics and chemistry. http://www.cs.berkeley.edu/~wkahan/SIAMjvnl.pdf, July 1997. URL http://www.cs.berkeley.edu/~wkahan/SIAMjvnl.pdf.

[101] S. Kamil, Cy Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1 –12, April 2010. doi: 10.1109/IPDPS.2010.5470421.

[102] T. Katagiri, K. Kise, H. Honda, and T. Yuba. FIBER: a generalized framework for auto-tuning software. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 146–159, 2003.

[103] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. ABCLibScript: a directive to support specification of an auto-tuning facility for numerical software. *Parallel Computing*, 32(1):92–112, November 2005. ISSN 0167-8191. doi: 10.1016/j.parco.2005.09.005. URL http://www.sciencedirect.com/science/article/B6V12-4HNYM8C-4/2/494cae4cead9dcc4ece8465c37f71029.

[104] Daniel S Katz and Neil P Chue Hong. Software citation in theory and practice. In *International Congress on Mathematical Software*, pages 289–296. Springer, 2018.

[105] Raphael 'kena' Poss. Machines are benchmarked by code, not algorithms. September 2013. URL https://arxiv.org/abs/1309.0534.

[106] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner. One-level storage system. *IRE Transactions of Electronic Computers*, April 1962. URL http://www.dcs.gla.ac.uk/~wpc/grcs/kilburn.pdf.

[107] Donald Knuth. *The Art of Computer Programming 3*. Addison-Wesley, 1997. ISBN 0-201-89685-0.

[108] Koji Build System. The koji build system. https://pagure.io/koji, 2014. URL https://pagure.io/koji.

[109] William Kramer and David Skinner. Consistent application performance at the exascale. *Int. J. High Perform. Comput. Appl.*, 23(4):392–394, November 2009. ISSN 1094-3420. doi: 10.1177/1094342009347700. URL http://dx.doi.org/10.1177/1094342009347700. Calls for methods of ensuring consistent application performance on large and complicated machines.

[110] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim.*, 2(2):165–198, June 2005. ISSN 1544-3566. doi: 10.1145/1071604.1071607. URL http://doi.acm.org/10.1145/1071604.1071607.

[111] Laurence Berkeley National Lab. Singularity. http://singularity.lbl.gov/, 2015. URL http://singularity.lbl.gov/.

[112] LAPACK. LAPACK. http://www.netlib.org/lapack/, 1992. URL http://netlib.org/lapack/.

[113] Hewlett Packard Enterprise Logan Sankaran. Hpc clusters: Best practices and performance study. https://www.intel.com/content/dam/www/public/us/en/documents/presentation/hpc-clusters-best-practices-performance-study.pdf, November 2016. URL https://www.intel.com/content/dam/www/public/us/en/documents/presentation/hpc-clusters-best-practices-performance-study.pdf.

[114] David Lundstrom. *A few good men from UNIVAC*. MIT Press, July 1987. ISBN 9780262121200.

[115] et al M O'Boyle. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, April 2007. doi: 10.1109/CGO.2007.32. URL http://ieeexplore.ieee.org/servlet/opac?punumber=4145089.

[116] Eric Zhiqiang Ma. MRCC: A distributed C compiler system on MapReduce. https://www.ericzma.com/projects/mrcc/, 2017. URL https://www.ericzma.com/projects/mrcc/.

[117] NetLib maintainers. BLAS. http://www.netlib.org/blas/, 2011. URL http://www.netlib.org/blas/.

[118] Art Manion and Will Dormann. Cpu hardware vulnerable to side-channel attacks: Vulnerability note VU 584653. https://www.kb.cert.org/vuls/id/584653/, January 2018. URL https://www.kb.cert.org/vuls/id/584653/.

[119] P. D. V. Mann and U. Mittaly. Handling OS jitter on multicore multithreaded systems. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009. doi: 10.1109/IPDPS.2009.5161046.

[120] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998. ISSN 1049-3301. doi: 10.1145/272991.272995. URL http://doi.acm.org/10.1145/272991.272995.

[121] Peter Medawar. *Advice to a young scientist*. Harper and Row, New York, 1979. "Most scientists do *not* know how to write"; "very many scientists are not intellectuals"; "clarity has been achieved and the style, if not graceful, is at least not raw and angular.".

[122] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. TOP500 supercomputer sites, 1993. URL http://www.top500.org/.

[123] Dharmendra S. Modha. IBM research: Brain-inspired chip. https://www.research.ibm.com/articles/brain-chip.shtml, 2011. URL https://www.research.ibm.com/articles/brain-chip.shtml.

[124] Luca Mussi, Youssef S.G. Nashed, and Stefano Cagnoni. GPU-based asynchronous particle swarm optimization. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1555–1562, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0557-0. doi: 10.1145/2001576.2001786. URL http://doi.acm.org/10.1145/2001576.2001786.

[125] T Mytkowicz, A Diwan, M Hauswirth, and P F Sweeny. Producing wrong data without doing anything obviously wrong! *ACM SIGPLAN Notices*, 44:265–276, March 2009. ISSN 15232867. URL https://www.cis.upenn.edu/~cis501/papers/producing-wrong-data.pdf.

[126] NERSC. Shifter. https://github.com/NERSC/shifter, 2015. URL https://github.com/NERSC/shifter.

[127] Numerical Algorithms Group (NAG). E05 – Global Optimization of a Function. https://www.nag.co.uk/numeric/fl/nagdoc_fl23/html/E05/e05conts.html#E05, 2011. URL https://www.nag.co.uk/numeric/fl/nagdoc_fl23/html/E05/e05conts.html#E05.

[128] Numerical Algorithms Group (NAG). How to make best use of the AMD Interlagos processor. http://www.hector.ac.uk/cse/reports/interlagos_whitepaper.pdf, November 2011. URL http://www.hector.ac.uk/cse/reports/interlagos_whitepaper.pdf.

[129] Numerical Algorithms Group (NAG). NAG library manual, mark 23. https://www.nag.co.uk/numeric/fl/nagdoc_fl23/html/frontmatter/manconts.html, 2011. URL https://www.nag.co.uk/numeric/fl/nagdoc_fl23/html/frontmatter/manconts.html.

[130] James Pallister, Simon J. Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. *The Computer Journal*, 58(1):95–109, 2015. doi: 10.1093/comjnl/bxt129. URL http://comjnl.oxfordjournals.org/content/58/1/95.abstract.

[131] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 319–332, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: 10.1109/CGO.2006.38. URL `http://dx.doi.org/10.1109/CGO.2006.38`.

[132] Francois Panneton and Pierre L'Ecuyer. WELL random number generator. `http://www.iro.umontreal.ca/~panneton/WELLRNG.html`, 2005. URL `http://www.iro.umontreal.ca/~panneton/WELLRNG.html`.

[133] D Patterson, K Yellick, K Keutzer, and R Bodik. Dwarf mine. `http://view.eecs.berkeley.edu/wiki/Dwarf_Mine`, 2014. URL `http://view.eecs.berkeley.edu/wiki/Dwarf_Mine`.

[134] Magnus Erik Hvass Pedersen. Good parameters for particle swarm optimization. *Hvass Lab., Copenhagen, Denmark, Tech. Rep. HL1001*, 2010.

[135] Magnus Erik Hvass Pedersen and Andrew J Chipperfield. Simplifying particle swarm optimization. *Applied Soft Computing*, 10(2):618–628, 2010.

[136] Antoine Petitet, R. Clint Whaley, Jack Dongarra, and A. Cleary. HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. `http://www.netlib.org/benchmark/hpl/`, December 2008. URL `http://www.netlib.org/benchmark/hpl/`.

[137] Antoine Petitet, R. Clint Whaley, Jack Dongarra, and A. Cleary. HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers, 2008. URL `http://www.netlib.org/benchmark/hpl/`.

[138] Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. Automatic tuning of compiler optimizations and analysis of their impact. *Procedia Computer Science*, 18:1312 – 1321, 2013. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2013.05.298. URL `http://www.sciencedirect.com/science/article/pii/S1877050913004419`. 2013 International Conference on Computational Science.

[139] Martin Pool. DistCC: A free distributed C/C++ compiler system. `https://github.com/distcc/distcc`, 2002. URL `https://github.com/distcc/distcc`.

[140] Sebastian Pop and Richard Guenther. Re: [graphite] use params for the size of loop blocking tiles. `https://gcc.gnu.org/ml/gcc-patches/2009-02/msg00994.html`, 2009. URL `https://gcc.gnu.org/ml/gcc-patches/2009-02/msg00994.html`.

[141] Mihail Popov, Chadi Akel, William Jalby, and Pablo de Oliveira Castro. Piecewise holistic autotuning of compiler and runtime parameters. In Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2016 Parallel Processing - 22nd International Conference*, volume 9833 of *Lecture Notes in Computer Science*, pages 238–250. Springer, 2016. ISBN 978-3-319-43659-3.

[142] POSIX Standard. IEEE Std 1003.1-2017 (revision of IEEE Std 1003.1-2008) - IEEE standard for information technology–portable operating system interface (POSIX®)

base specifications, issue 7. https://standards.ieee.org/findstds/standard/1003.1-2017.html, 2017. URL https://standards.ieee.org/findstds/standard/1003.1-2017.html.

[143] Steven Przybylski. *Cache and memory hierarchy design: a performance-directed approach*. Morgan Kaufmann, 1990.

[144] Szilárd Páll and Berk Hess. A flexible algorithm for calculating pair interactions on SIMD architectures. *Computer Physics Communications*, 184(12):2641–2650, 2013. ISSN 0010-4655. doi: 10.1016/j.cpc.2013.06.003. URL http://www.sciencedirect.com/science/article/pii/S0010465513001975.

[145] Shah Faizur Rahman, Jichi Guo, and Qing Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 107–116, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0241-8. doi: 10.1145/1944862.1944880. URL http://doi.acm.org/10.1145/1944862.1944880.

[146] Ari Rasch and Sergei Gorlatch. ATF: A generic directive-based auto-tuning framework. *Concurrency and Computation: Practice and Experience*, 2017. ISSN 1532-0634. doi: 10.1002/cpe.4423. URL http://dx.doi.org/10.1002/cpe.4423. e4423 cpe.4423.

[147] Kaz Sato, Cliff Young, and David Patterson. An in-depth look at Google's first tensor processing unit (TPU). https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, 2017. URL https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[148] Joseph Schuchart, Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Ramkumar Nagappan, and Michael K. Patterson. The shift from processor power consumption to performance variations: fundamental implications at scale. *Computer Science - Research and Development*, 31(4):197–205, Nov 2016. ISSN 1865-2042. doi: 10.1007/s00450-016-0327-2. URL https://doi.org/10.1007/s00450-016-0327-2.

[149] M. Schumer and K. Steiglitz. Adaptive step size random search. *IEEE Transactions on Automatic Control*, 13(3):270–276, June 1968. ISSN 0018-9286. doi: 10.1109/TAC.1968.1098903.

[150] Jeffrey J. Schutkoske. Cray XC system node level diagnosability. CUG '15. Cray User Group, 2015. URL https://cug.org/proceedings/cug2015_proceedings/includes/files/pap130.pdf.

[151] J. F. Schutte, J. A. Reinbolt, B. J. Fregly, R. T. Haftka, and A. D. George. Parallel global optimization with the particle swarm algorithm. *International journal for numerical methods in engineering*, 61(13):2296–2315, December 2004. doi: 10.1002/nme.1149. URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1989676. PMC1989676.

[152] K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *Cluster Computing, 2008 IEEE International Conference on*, pages 421–429, Sept 2008. doi: 10.1109/CLUSTR.2008.4663803.

[153] Jeff Squyres. Ticket #1244: Warn when system() or fork() is used with OFED-based BTLs. `https://svn.open-mpi.org/trac/ompi/ticket/1244`, March 2008. URL `https://svn.open-mpi.org/trac/ompi/ticket/1244`.

[154] Richard M Stallman. GNU compiler collection internals. 1987. URL `http://gcc.gnu.org`.

[155] Standard Performance Evaluation Corporation. SPEC's benchmarks. `https://spec.org/benchmarks.html`, 2018. URL `https://spec.org/benchmarks.html`.

[156] Charles Stross. *TOAST*. Self, 2002. ISBN 0-8095-5603-0. URL `http://www.antipope.org/charlie/blog-static/fiction/toast/toast-intro.html`.

[157] SUSE. Icecream: Distributed compiler with a central scheduler to share build load. `https://github.com/icecc/icecream`, 2012. URL `https://github.com/icecc/icecream`.

[158] The Chicken Team. `https://www.call-cc.org`, 2000. URL `https://www.call-cc.org`.

[159] The Glasgow Haskell Team. The glasgow haskell compiler (GHC). `https://www.haskell.org/ghc/`, 1997. URL `https://www.haskell.org/ghc/`.

[160] The LLVM Team. The LLVM compiler infrastructure project. `http://llvm.org/`, September 2009. URL `http://llvm.org/`.

[161] The OpenMPI Team. OpenMPI: A high performance message passing library. `https://www.open-mpi.org/`, 2004. URL `https://www.open-mpi.org/`.

[162] The OpenMPI Team. OpenMPI FAQ: Fork warning. `https://www.open-mpi.org/faq/?category=tuning#fork-warning`, 2017. URL `https://www.open-mpi.org/faq/?category=tuning#fork-warning`.

[163] The Racket Team. Racket scheme. `https://racket-lang.org/`, 1995. URL `https://racket-lang.org/`.

[164] The GW4 Alliance. GW4 isambard. `http://gw4.ac.uk/isambard/`, 2017. URL `http://gw4.ac.uk/isambard/`.

[165] Ilian T Todorov, William Smith, Kostya Trachenko, and Martin T Dove. DL_POLY_3: new dimensions in molecular dynamics simulations via massive parallelism. *Journal of Materials Chemistry*, 16(20):1911–1918, 2006.

[166] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I August. Compiler optimization-space exploration. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 204–215, March 2003. doi: 10.1109/CGO.2003.1191546.

[167] Robert Van De Geijn. The GotoBLAS/BLIS approach to optimizing matrix-matrix multiplication - step-by-step. https://github.com/flame/how-to-optimize-gemm/wiki, 2014.

[168] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. Parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. *JACIC*, 3(3):123–137, 2006.

[169] R. Clint Whaley. User contribution to ATLAS. http://math-atlas.sourceforge.net/devel/atlas_contrib/, 2007. URL http://math-atlas.sourceforge.net/devel/atlas_contrib/.

[170] R. Clint Whaley. Re: Atlas contribution documentation. Communication to J.R.Jones, 2009, February 2009.

[171] R. Clint Whaley. Improving flag selection using mmflagsearch. http://math-atlas.sourceforge.net/atlas_install/node20.html, 2010. URL http://math-atlas.sourceforge.net/atlas_install/node20.html. Mentioned in the changelog: "ATLAS 3.9.23 released 02/07/10, changes from 3.9.22:" .. " * Added a compiler flag search to ease job of finding good flags. - ATLAS/tune/blas/gemm/mmflagsearch.c"
.

[172] R. Clint Whaley. Re: [atlas-devel] MIC. https://sourceforge.net/p/math-atlas/mailman/message/31537905/, October 2013.

[173] R. Clint Whaley. Re: Atlas contribution documentation. Communication to J.R.Jones, 2014, January 2014.

[174] R Clint Whaley. Rough guide to overriding ATLAS's compiler choice/changing flags. http://math-atlas.sourceforge.net/atlas_install/node17.html#sec-cc-override, 2016. URL http://math-atlas.sourceforge.net/atlas_install/node17.html#sec-cc-override.

[175] R. Clint Whaley. #1029 support for Broadwell-EP processors? https://sourceforge.net/p/math-atlas/support-requests/1029/, July 2017.

[176] R. Clint Whaley and Anthony. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Software: Practice and Experience*, 2008. URL http://www.csc.lsu.edu/~whaley/papers/timing_SPE08.pdf.

[177] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2): 101–121, 2005. doi: 10.1002/spe.626. URL http://dx.doi.org/10.1002/spe.626.

[178] R. Clint Whaley, Jack J Dongarra, and Antoine Petitet. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, January 2001. doi: 10.1016/S0167-8191(00)00087-9.

[179] Maurice V Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, (2):270–271, April 1965.

[180] Zhang Xianyi, Wang Quian, Zaheer Chothia, Chen Shaohu, Luo Wen, Stefan Karpinski, and Mike Nolta. OpenBLAS, March 2011. URL http://xianyi.github.com/OpenBLAS/.

[181] Xu Yang, John Jenkins, Misbah Mubarak, Robert B Ross, and Zhiling Lan. Watch out for the bully! job interference study on dragonfly network. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 750–760. IEEE, 2016.

[182] Tao Ye and Shivkumar Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 196–205, New York, NY, USA, 2003. ACM. ISBN 1-58113-664-1. doi: 10.1145/781027.781052. URL http://doi.acm.org/10.1145/781027.781052.

[183] Xianyi Zhang. OpenBLAS FAQ: What is OpenBLAS? why did you create this project? https://github.com/xianyi/OpenBLAS/wiki/faq#what, December 2011. URL https://github.com/xianyi/OpenBLAS/wiki/faq#what.

[184] Xianyi Zhang. Please support AMD Bulldozer #118. https://github.com/xianyi/OpenBLAS/issues/118, June 2012.