Imperial College London
Department of Electrical and Electronic Engineering

***Portable, Predictable and Partitionable*: A Domain Specific Approach to Heterogeneous Computing**

Gordon Eric Inggs

December, 2015

Supervised by Dr David B. Thomas and Prof Wayne Luk

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Engineering of Imperial College London
and the Diploma of Imperial College London

Abstract

Computing is increasingly heterogeneous. Beyond Central Processing Units (CPUs), different architectures such as massively parallel Graphics Processing Units (GPUs) and reconfigurable Field Programmable Gate Arrays (FPGAs) are seeing widespread adoption. However, the failure of conventional programming approaches to support portable execution, predict the runtime characteristics and partition workloads optimally is hindering the realisation of heterogeneous computing.

By narrowing the scope of expression in a natural manner, using a domain specific approach, these three challenges can be addressed. A domain specific heterogeneous computing methodology enables three features: *Portability*, *Prediction* and *Partitioning*. Portable, efficient execution is enabled by a domain specific approach because only a subset of domain functions need to be supported across the heterogeneous computing platforms. Predictive models of runtime characteristics are enabled as the structure of the domain functions may be analysed *a priori*. Finally optimal partitioning is possible because the metric models can be used to form an optimisation program that can be solved by either heuristic, machine learning or Mixed Integer Linear Programming (MILP) approaches.

Using the example of the application domain of financial derivatives pricing, a domain specific application framework, the Forward Financial Framework (F^3), can execute a single pricing task upon a diverse range of CPU, GPU and FPGA platforms from many different vendors. Not only do these portable implementations exhibit strong parallel scaling, but are competitive with state-of-the-art, expert created implementations of the same option pricing problems. Furthermore, F^3 can model the crucial runtime metrics of latency and accuracy for these heterogeneous platforms using a small benchmarking procedure to within 10% of the run-time value of these metrics. Finally, the framework can optimally partition work across heterogeneous platforms, using a MILP framework, that is up to 270 times more efficient than what is achieved by using a heuristic approach.

Declarations

Declaration of Originality

I, Gordon Eric Inggs, declare that all of the work in this dissertation is either my own or appropriately referenced.

Declaration of Copyright

Publications

I have been lead author on the following publication that were based upon elements of this dissertation.

Peer-Reviewed

Published

- "Bringing Heterogeneous Computing to the End User", Proceedings of Psychology of Programming Interest Group (PPIG) Work-in-Progress Workshop, 2013.
- "A Heterogeneous Computing Framework for Computational Finance", Proceedings of the International Conference on Parallel Processing (ICPP), 2013.
- "A Domain Specific Approach to Heterogeneous Computing: From Availability to Accessibility", Proceedings of the International Workshop on FPGAs for Software Programmers (FSP), 2014.
- "Is high level synthesis ready for business? A computational finance case study", Proceedings of the International Conference on Field-Programmable Technology (FPT), 2014.
- "Exascale Computing for Everyone: Cloud-based, Distributed and Heterogeneous", Proceedings of the Exascale Applications and Software Conference (EASC), 2015.
- "Seeing Shapes in Clouds: On the Performance-Cost trade-off for Heterogeneous Infrastructure-as-a-Service", Proceedings of the International Workshop on FPGAs for Software Programmers (FSP), 2015.

Under-Review

- "An Efficient, Automatic Approach to High Performance Heterogeneous Computing", IEEE Transactions on Parallel and Distributed Systems (TPDS), 2015

Invited

- "Is Altera's OpenCL SDK ready for business?", White Paper, Nallatech Inc., 2014
- "Is High Level Synthesis ready for business? An Option Pricing Case Study", Chapter, FPGA Based Accelerators for Financial Applications, 2015

Acknowledgements

This dissertation would not be possible without the material means to undertake it, and so I would like to gratefully acknowledge the funding support I have received from the United Bank of Switzerland, the Department of Electrical and Electronic Engineering at Imperial College London and Goodenough College. I would particularly like to thank the Oppenheimer Memorial Trust and the South African National Research Foundation, who, despite being South Africa-based and on the wrong end of unfavourable exchange rates, contributed significant proportions of the funding that supported me.

I would like to thank Jill Hodges and FireTech Camp, as well as the Departments of Computing, and Electrical and Electronic Engineering at Imperial College London for the opportunity to undertake teaching during my studies. Beyond the material necessity of this work, teaching gave me desperately needed perspective on my research, and computer engineering more broadly. For the last six months at Imperial, I have held a Research Assistantship under Prof George Constantinides, which has been an invaluable opportunity to immerse myself further in fascinating research, and for this opportunity I am grateful.

I am also grateful to Maxeler, Nallatech, Altera, Xilinx, the Radar and Remote Sensing Group at the University of Cape Town and Imperial College London for providing access to software and equipment that were instrumental to my research.

Moving to less tangible forms of support, I have received invaluable advice over the course of my research from Prof Nachiket Kapre at Nangyang Technological University and Prof George Constantinides. I would also like to thank Dr Christos Bouganis at Imperial College London who served as both my early stage assessor, and late stage reviewer. Dr Bouganis's comments and words of encouragement came at critical moments in my work. Wiesia Hsissen must be one of the most capable administrators on the planet, and I am indebted for her advice and assistance in navigating Imperial's bureaucracy.

I was lucky enough to be a member of two research groups, the Custom Computing Group in the Department of Computing, and the Circuits and Systems in the Department of Electrical and Electronic Engineering. I will sorely miss the collegiality of these groups, particularly the many stimulating discussions. I can recall conversations with Dr Sam Bayliss, Dr Theo Drane, Dr Qiwei Jin, Dr David Boland, Dr Eddie Hung, Dr Pavel Burovskiy, Ms Joan Omeru, Mr Aryan Tavakkoli, Dr Andrea Suardi and Dr Eric Kerrigan to name but a few that have had a direct benefit to my research. Finally, I have had the immense privilege of collaborating with Mr Andrea Picciau, Mr Peter Ogdan, Mr Andrew Bean and Mr Shane Fleming, whose patience, good humour and wit is only matched by their competencies.

Prof Wayne Luk was the reason that I applied to Imperial College London, and the quality of advice that I have received from him during my research has vindicated my rationale. Working with a scholar of such standing has been a great honour. Amongst the many significant contributions that Prof Luk made to my work was suggesting that Dr David B. Thomas serve as my primary supervisor.

Dr Thomas is truly a Scholar for the 21st century. While I am regularly struck by the extraordinary scale of his engineering ability and critical insight, I have benefited the most from his humble approach. David makes little distinction between teaching and learning, and in doing so he has shown me how kindness, curiosity and rigour can be fused into a vocation, and for this, I will always be in his debt.

The support of my friends, some of whom have already been mentioned above, has been invaluable to me throughout my PhD. They put up with unreasonably detailed conversations on the minutiae of heterogeneous computing, encouraged me when I've flagged, and when necessary, braaied with me in the London rain. The very best that I can aspire to is being as good a friend as those that I have. A significant number of my London friends are members of Goodnough College, a community that has truly been home for the past three years.

My family, near and far, have been a bottomless well of support and encouragement. Despite the often great distances separating us, I am always aware of their deep love and almost fanatical belief in my abilities. My parents, Mike and Trish, came to London several decades ago, and so truly blazed the trail which I have merely followed in. I am under no illusion that the last two decades of study would have been even remotely possible without my family.

Rebecca, my partner in every sense of the word, moved to London and completed a PhD during the same time period that I have. Her unconditional love and support, infused with her unique clarity of thought, saw, and indeed often carried me through this undertaking. For this, and many more reasons, she has my love.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Introduction

Computing is becoming increasingly heterogeneous. Over the last decade both the fundamental composition and organisation of computing systems has become more diverse. This diversification is in response to challenges to the hitherto dominant approach to general-purpose computing. These challenges include the memory and power "walls", as well as the increasing viability of alternatives such as massively parallel Graphics Processing Units (GPUs), and custom accelerator architectures implemented using reconfigurable Field Programmable Gate Arrays (FPGAs).

Different architectures beyond the Von Neumann have risen to prominence, and have been combined in various permutations with a large variety of interconnection technologies. Infrastructure-as-a-Service or Utility computing is also coming to the fore, with the physical and conceptual separation between computing hardware and programmer growing.

This explosion of diversity, or rather the factors driving it, are often described as threats or crises[1] in the literature of computer systems engineering. I argue in this dissertation that these are in fact opportunities. In meeting these challenges, as an engineering discipline we have the chance to formulate more general theories of computing; develop technologies that are more flexible and suited to the specific requirements of users; however, most importantly in my opinion, the opportunity exists to make the field fundamentally more accessible.

In this dissertation I provide a means to realise the opportunity of heterogeneous computing. In this chapter, I define and elaborate on what I believe are three critical features required for greater accessibility: *Portability*, *Predication* and *Partitioning*, as well as a methodology, based upon domain specific computing, for implementing these features.

This chapter continues by placing this dissertation in the context of the heterogeneous computing by defining heterogeneous computing, and then outlining the chief challenge and opportunity in this area. I then describe the three features of the research problem which are necessary to address this challenge and realise this opportunity, and the three research questions that must be answered in doing so. After this, I consider the impact of this project in terms of contributions made, in terms of theoretical impact and practical experience. Finally, I present an overview of the rest of the dissertation, describing the structure of its arguments.

[1]See Sutter [7], Hartenstein [8] or Moore [9] for examples of this interpretation

1.2. The Opportunity and Challenge of Heterogeneous Computing

Heterogeneous Computers are computing systems that are comprised of two or more architecturally distinct computing platforms, such as multicore CPUs, GPUs and FPGAs. Increasingly commodity computing systems are comprised of several platforms, as the power and memory walls continue to limit the capabilities of CPUs, while the smaller process technologies that keep pace with Moore's law result in alternatives, such as GPUs and FPGAs, that continue to improve in functionality and efficiency.

Heterogeneous computing broadly describes these new computing systems, composed of multiple subsystems. As a result, heterogeneous computing technologies are reliant upon parallel execution for improved performance, evaluating components of the problem concurrently. This new multi-architectural, parallel paradigm offers opportunities, but also poses challenges.

1.2.1. The Challenge - Amdahl's Law

The most prominent challenge in using heterogeneous computing systems is the same as that faced by all cooperative endeavours - division of labour.

This challenge was captured by Amdahl [10] in the optimisation of computational latency as given in (1.1), where p is the degree of parallel, homogeneous resources available, e is the sequential component of the task and $S(p)$ the speedup of the parallel implementation over execution upon a single unit of the compute resource, as a function of the degree of parallelism.

$$S(p) = \frac{1}{e + \frac{1}{p}(1-e)} \tag{1.1}$$

Amdahl's argument is that even with infinite parallel computing capability, the acceleration of a computational task will always be bounded by those operations which cannot be computed in parallel, as expressed in (1.2).

$$\lim_{p\to\infty} \quad S = \frac{1}{e} \tag{1.2}$$

The resolution of the challenge posed by Amdahl is in reformulating computational tasks such that e, the sequential component, is minimised, and p, the parallel, maximised. The benefit of the reformulation has to be balanced with any cost of doing so, such as an increase in the degree of synchronisation communication required.

A further consideration is that Amdahl's formulation assumes idealised, uniform computing resources. Heterogeneous systems add an additional level of implementation complexity, as a potentially unique implementation has to be provided for each platform. Furthermore, because of the inconsistent capabilities of platforms, not only does the best platform for the sequential component have to be chosen, but also an optimal division of work for the parallel component has to be found.

Often the increased complexity is so great that applications ostensibly execute more efficiently on a homogeneous architecture rather than a heterogeneous one, even when the heterogeneous configuration has more computing power, as shown in the experiments that I describe in this dissertation, as well as other contexts such as high performance computing clusters [11]. This can even be case when the heterogeneous configuration apparently have more computing resources than the homogeneous case. This underperformance is almost always due to the increased complexity of balancing the relative capabilities of the heterogeneous system, arising from the interaction between user application, architecture and programming tool.

1.2.2. The Opportunity - Super-linear Performance Scaling

While the previous section described Amdahl's challenge in using parallel computing systems, there is an opportunity in the inconsistency of heterogeneous systems. This opportunity is *super-linear* performance scaling. Super-linear scaling is where the performance of the combined heterogeneous resources exceeds the sum of the individual platforms' performances. Such scaling can only be achieved if there are inconsistent capabilities between platforms, i.e. some tasks execute more efficiently relative to others upon certain architectures, as is often the case with heterogeneous platforms.

For example, given two computational tasks, one that is a simple loop with a large amount of arithmetic computation in its body, and the other task comprised mainly of unbalanced conditional statements. The first task relative to the second will execute disproportionately more efficiently on a GPU, whilst the inverse would most likely be true for a Multicore CPU.

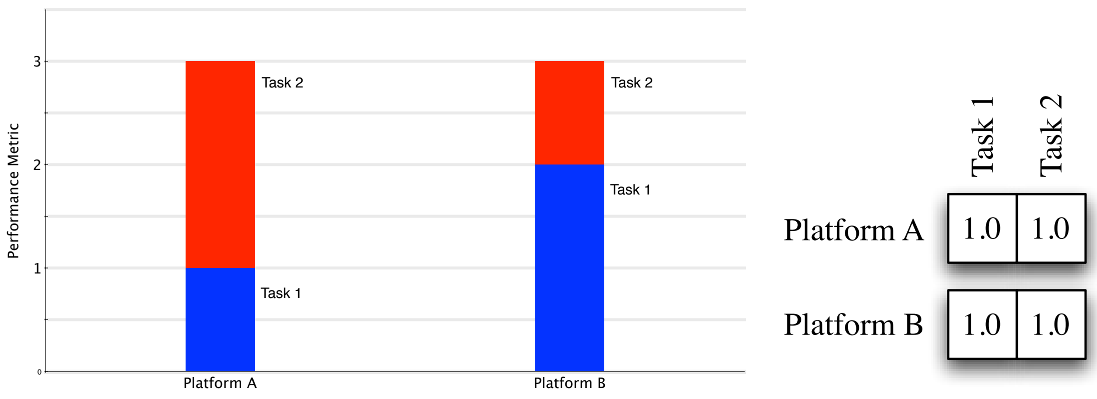
I have illustrated the concept of super-linear scaling in Figure 1.1 using a hypothetical example. Firstly, the independent execution of a workload of two divisible tasks upon two platforms is illustrated in Figure 1.1a. In the second subfigure, Figure 1.1b, the workload is balanced across the two platforms equally, so that each is performing half of each task. In this balanced case, linear scaling is achieved, as using both platforms results in a performance improvement in proportion to the total capabilities of the platforms, as measured by the sum of these two tasks. However, if an allocation that exploits inconsistent platform capabilities is used, as given in Figure 1.1c, super-linear performance scaling is achieved. Each platform is matched with the tasks which it performs best, and hence the total performance exceeds the sum of the independent platform capabilities.

1.3. Research Problem

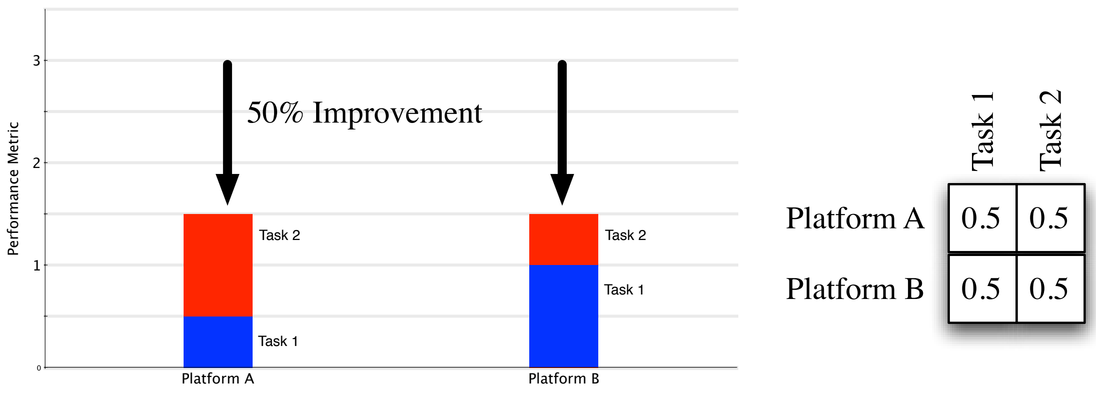
In the previous section I discussed the major challenge to heterogeneous computing, as identified by Amdahl, and the opportunity of super-linear performance scaling. In this section, I now describe the three features of the research problem posed by this challenge and opportunity: Portability, Predictability and Partitionability.

1.3.1. Portability

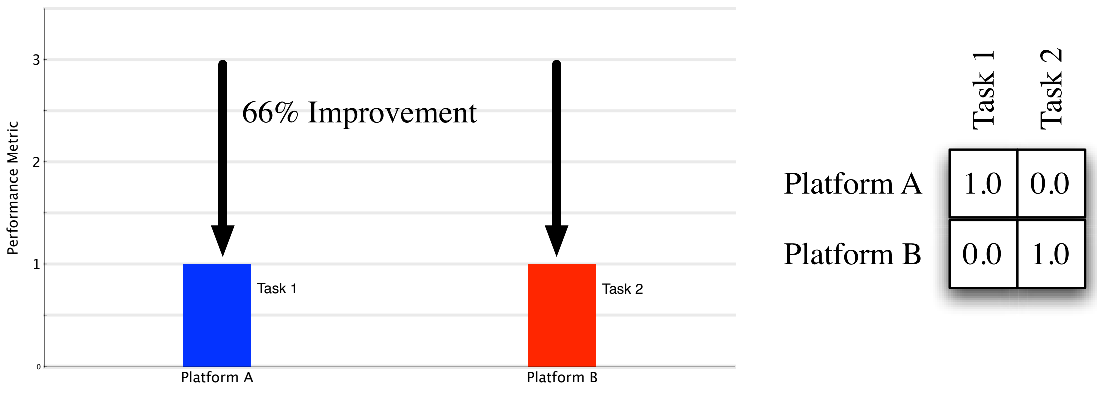
Implicit in considering the use of heterogeneous computing platforms is the assumption that programmers have a way to use these platforms. Ideally a single mechanism, such as a programming standard or tool, exists that could be used to program architecturally different platforms. I define such a mechanism as being functionally portable.



(a) Both tasks running on both platforms, with identical performance.



(b) Tasks balanced equally across platforms, achieving a linear performance improvement.



(c) Tasks matched to platforms, achieving a super-linear performance improvement.

Figure 1.1.: Illustration of the potential for super-linear performance scaling using two tasks and two platforms. A lower performance metric score is better. The matrix to right of each figure is the allocation matrix, where the *(i,j)* entry gives the allocation of task *j* to platform *i*.

However, a further concern is the depth of knowledge required to use a specific platform to the full extent of its capabilities. If a programmer with limited understanding of the platform is nevertheless able to make full use of it, then I define such an approach as being efficiently portable.

1.3.2. Predictability

Beyond being able to implement tasks efficiently, programmers need to be able to predict how their tasks will run on different architectures. Independent of other considerations, programmers need these predictions to be able to balance their different objectives, such as performance requirements and resource constraints. The ability to predict the nature of a task implementation gives the programmer insight into the platform, and allows them to reason about how they might use it.

However, beyond balancing objectives, being able to predict how a task will run on different platforms enables the programmer to compare heterogeneous platforms prior to execution. Doing so is the first step in identifying how a group of tasks should be mapped to a group of heterogeneous platforms.

1.3.3. Partitionability

Given the ability to implement tasks across a range of platforms, and to predict the nature of these implementations, a further feature is still required to achieve the performance predicted by Amdahl. That feature is the means to partition work across the available resources in proportion to the relative capabilities of those platforms. This then allows for the performance improvement given in (1.1), where the value of p is the combined, relative parallel compute capability of the available platforms.

However, to go beyond the performance limit identified by Amdahl, a partitioning approach is required that is able to exploit inconsistencies across tasks and platforms. A partitioning approach that is able to do so, that matches tasks to the best possible platform, allows for super-linear performance to be achieved, as illustrated in Figure 1.1.

1.4. Research Questions

There are three questions that this dissertation must answer so as to address the features of the research problem outlined in Section 1.3:

1. Is it feasible to support the execution of a single computational task description upon diverse, heterogeneous computing systems? Beyond the capability to do so, can such an execution be efficiently portable, i.e. using a significant degree of the target platform's compute capability?

2. Can the characteristics of tasks be modelled across heterogeneous computing systems so that performance is predictable? Is there an abstraction for doing so, such that these characteristics are meaningful to the programmer without requiring architectural knowledge?

3. Can tasks be made partitionable across heterogeneous computing systems with efficiency such that programmer objectives are balanced while taking full advantage of the differing capabilities of the platforms?

1.5. Contributions

My dissertation demonstrates how the heterogeneous computing research problem that I have outlined can be addressed using a domain specific methodology. The workflow that I envision programmers following is illustrated in Figure 1.2.

To realise this methodology, I had to answer the research questions outlined in Section 1.4. I will show how the first two research questions, and hence features, portability and predictability, can be be addressed through automatic means using a domain-specific methodology. The third question and corresponding feature, partitionability, I argue can only be addressed in cooperation with the programmer. Ultimately only the programmer can balance competing objectives in the face of the complex interaction of application and computing resources. However, I show that a domain specific flow can provide a representation of the design space that would be intuitively understandable to the programmer, using knowledge embedded in the application domain.

In implementing the domain specific approach, I have made both practical and theoretical contributions.

1.5.1. Practical

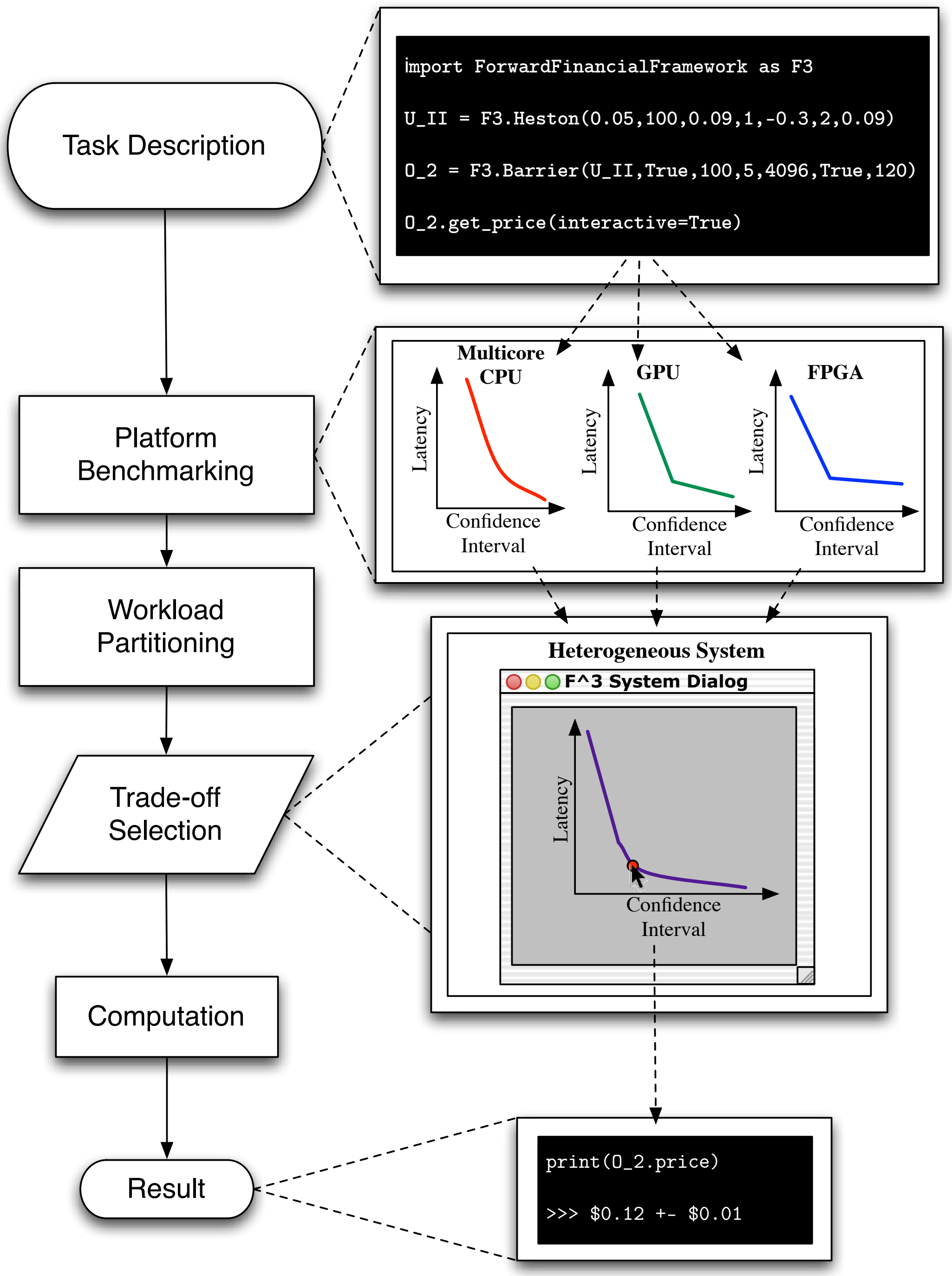
The tangible artefacts I have created in addressing these questions:

- *Portability*: A domain specific, heterogeneous computing framework for computational finance, the Forward Financial Framework (F^3)[2]. The framework currently supports the execution of a large subset of option pricing tasks upon x86 and ARM multicore CPUs using POSIX Threads; Nvidia and AMD GPUs as well as Intel's Xeon Phi using OpenCL; Xilinx and Altera FPGA platforms using the Xilinx, Altera and Maxeler's programming tools. Further details of F^3 are given in Chapter 5.

- *Prediction*: Models for estimating the latency and accuracy run-time characteristics of financial option pricing tasks using a short online benchmarking procedure. Using F^3, prediction models for financial problem metrics such as latency, accuracy and financial cost have been formulated and verified for the heterogeneous computing platforms supported. Further details on latency and accuracy models may be found in Chapter 6.

- *Partitioning*: Generation of efficient allocations of work. Using the performance models described above as well as global optimisation algorithms and Mixed Integer Linear Programming (MILP) techniques, representations of the design space for financial problems can be created that convey the range of performance possible for the computational resources available. More information can be found in Chapter 7.

[2]The source code for the framework may be found at https://github.com/Gordonei/ForwardFinancialFramework



```
import ForwardFinancialFramework as F3

U_II = F3.Heston(0.05,100,0.09,1,-0.3,2,0.09)

O_2 = F3.Barrier(U_II,True,100,5,4096,True,120)

O_2.get_price(interactive=True)
```

```
print(O_2.price)

>>> $0.12 +- $0.01
```

Figure 1.2.: Proposed solution to research problem, illustrated using the Forward Financial Framework (F^3)

1.5.2. Theoretical

The dissertation makes the following conceptual contributions:

- Broad definition of heterogeneity - I promote a broad definition of heterogeneity in computing, while still performing computational tasks found in active application domains. Such an approach improves the general applicability of the field.
- Insight into the components of computing - a better understanding of the interplay between application domain, means of expression, platform and user. I unify these factors in design space abstractions that allow the programmer to make trade-offs that they understand.
- Balance between the application domain and implementation platforms - a more robust model for computing is encouraged through this research, one that is not overly dominated by the application domain or the the platform(s) of implementation, yet responsive and inclusive of a broad range of both.
- Suggestions for a general model for domain specific heterogeneous computing - the culmination of this dissertation is recommendations for mapping a problem expressed in terms of the language of its domain to a broadly heterogeneous computing system.

1.5.3. Scope

I now define the scope of this project, considering what areas it will, and will not cover.

Balancing Theory and Practice

I have defined the programming methodology that I describe unambiguously, using mathematical formalisms. However, this formalisation is not where the majority of the contributions of the dissertation lie. Rather it serves as a clarification of meaning as opposed to a complete mathematical proof of the domain specific methodology. As a result, I have rather demonstrated these principles in action as opposed to enumerating the full extent of the formalism.

Case Study as Evidence

Related to the previous point, I have made use of a single, practical case study to lend weight to the claims that I make with regards to a domain specific methodology. A single instance doesn't constitute proof by scientific standards, i.e. that these features exist in every application domain and will be useful. Its existence in one domain though is significant, as given the domain agnostic nature of the methodology, the fact that it works for the given domain is an encouraging sign that it can be applied to others.

1.6. Overview of Dissertation

In Figure 1.3, I have provided the logical layout of the dissertation.

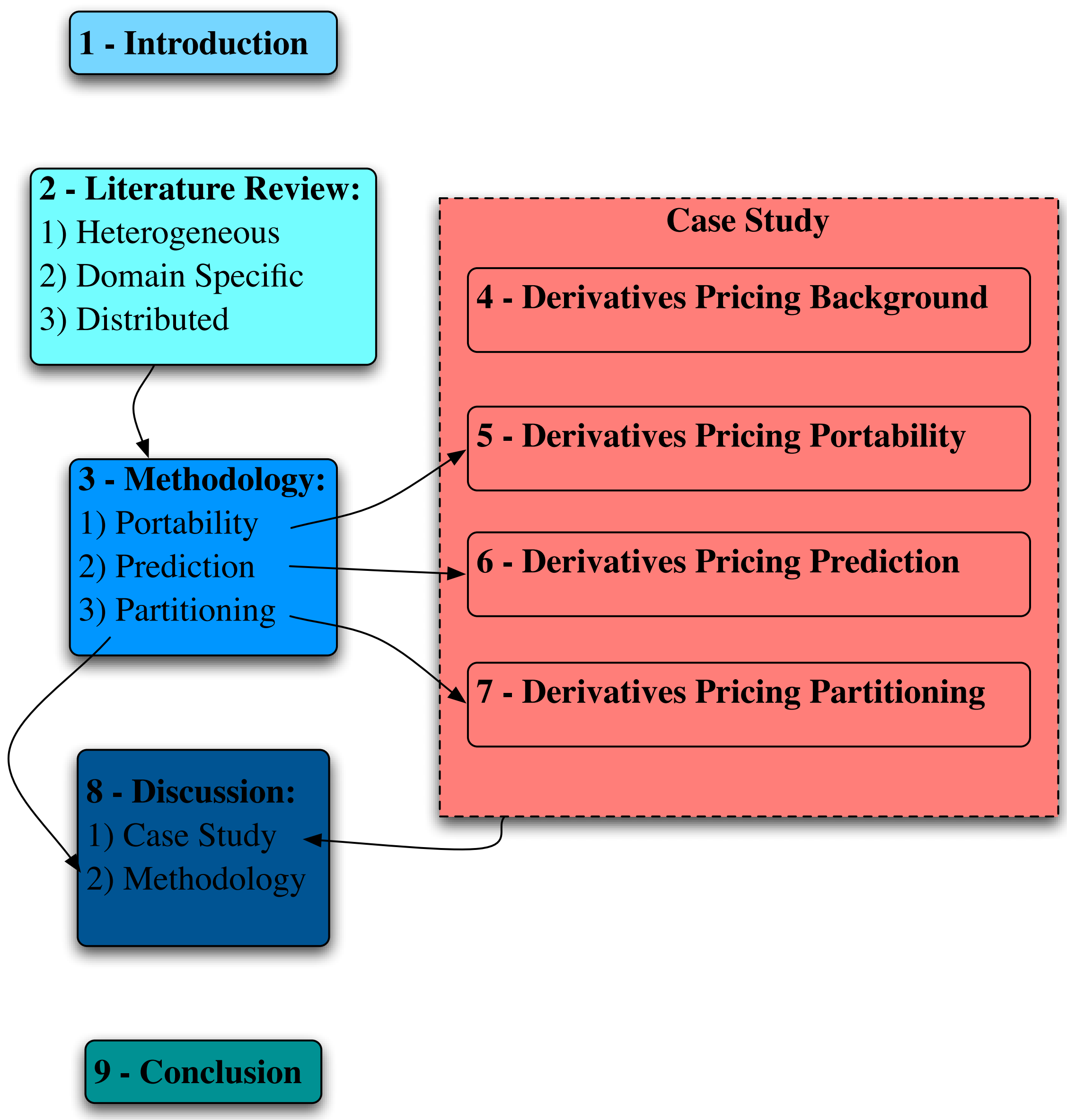


Figure 1.3.: Overview of Dissertation

1.6.1. Part I - the Background

In the first part of this dissertation I discuss the my understanding of the current state of the art, as well as outline the domain specific methodology that I propose.

The chapter that follows is a review of a subsection of the research literature in Heterogeneous, Domain Specific and Distributed Computing and the intersections between the three. The Literature Review chapter concludes by evaluating the key themes identified, and the implications for addressing the research questions.

Chapter 3 details the domain specific methodology that I propose, drawing upon the insights derived from the literature. I define formally each of the three features that this methodology enables, namely portable execution, predictive modelling and the partitioning of workloads. In addition to describing the supporting analysis for each feature, I provide the falsifiable criteria upon which the existence of these features can be assessed.

1.6.2. Part II - the Case Study

The second part of the dissertation is concerned with the computational finance case study that I use to evaluate the domain specific methodology that I propose. In addition to describing the background to the derivatives pricing computational domain, I describe how each of the three domain specific features can be achieved. I also evaluate each, testing the fulfilment of the criteria described in Chapter 3.

The demonstration application domain of Computational Finance is defined in Chapter 4. Firstly, computational finance problems are described in detail, in terms of the financial products that are being priced as well as the Monte Carlo Pricing algorithm used in this project. The computational finance problems are then described as a computational domain, as defined in the domain specific literature discussed in Chapter 2.

In Chapters 5, 6 and 7, I evaluate the portability, prediction and partitioning features are implemented within the domain specific approach respectively. In addition to describing how each feature is realised for the derivatives pricing case study, I demonstrate with an experimental evaluation how the respective criteria have been satisfied. In all three cases I have used F^3 to demonstrate and evaluate the features under consideration.

The outcome of the case study is a practical demonstration of how the domain specific methodology can not only make heterogeneous computing accessible, but also enables super linear performance scaling.

1.6.3. Part III - the Analysis

In the final part, I analyse the methodology and the case study that I have undertaken to verify it. I first consider the limitations of both, and then conclude by considering the future directions this work could take.

In Chapter 8, I discuss the research that I have undertaken. I do so by considering the relationship between the case study and the domain specific methodology, and then the methodology more generally. In reflecting upon the case study and its relationship to the methodology, I first consider the degree to which the criteria identified in Chapter 3 have been fulfilled, and the limitations of the case study. In doing so, I argue that the case study is a valid instance of the domain specific methodology. I then consider criticisms of the methodology more broadly: the development effort required, the inherent assumptions, and finally, how the partitioning feature must be broadened to encompass scheduling.

Finally, I conclude this dissertation. I consider the degree to which the research questions outlined above have been addressed, as well as the scope for future work within this project. I then consider the broader experience of undertaking this project, and more general observations about computer engineering as a discipline.

Part I.

The Background

2. Literature Review

In this first part of the dissertation, I provide the background to my argument that a domain specific approach enables the effective use of heterogeneous computers. In this Literature Review, I consider the current state of the art of three relevant strands of computing research, and the implications for the implementation of the critical features of portability, prediction and partitioning. In Chapter 3, I describe the Domain Specific methodology that I propose for heterogeneous computing. In addition to providing a formal description of the elements of the method, I use two example domains to illustrate this methodology.

This chapter describes the background literature in terms of three distinct areas of computing that my project touches upon:

- **Heterogeneous computing**: different approaches to programming heterogeneous computing systems.
- **Domain specific computing**: a conception of computing limited to particular application area or domain.
- **Distributed computing**: the distribution of work to many computing platforms.

I conclude this review by analysing these areas with respect to the three key features required for broader access to heterogeneous computing, Portability, Prediction and Partitioning, highlighting the relevant considerations for my work.

2.1. Heterogeneous Computing

As defined in Section 1.2, heterogeneous computing is computing performed using two or more architecturally distinct computing devices. Also described in Chapter 1, was the potential of heterogeneous computing for super linear performance scaling, whereby the relative strengths of the available computing resources are exploited to achieve performance beyond the sum of the performance of these platforms.

In this subsection, I survey the prior art of heterogeneous computing. I begin by considering an approach to heterogeneous computing where each distinct computational device within the system is programmed independently. I then describe how the short-comings of this approach has prompted the emergence of heterogeneous computing standards, such as OpenCL, which enable the programming of multiple architectures using the same code. I conclude by considering general heterogeneous computing frameworks, which provide compilation and run-time support for more than one heterogeneous architecture, often by supporting a heterogeneous computing standard. I also evaluate the degree to which these frameworks address the challenges posed in programming heterogeneous systems.

2.1.1. Platform-specific Approaches

A common approach to programming heterogeneous computers is to use multiple, architecture-specific programming frameworks for the constituent platforms, the computing architectural instances, within the heterogeneous system, such as CPU compilers for multicore CPUs, and GPU and FPGA vendor supplied programming tools. While well-established and mature, the chief problem with such approaches is poor interoperability between different computational platforms.

Indeed, the more sophisticated and optimised the framework these tools provide, often the more platform-specific the required task code becomes. An extreme example of this is the Hardware Description Language (HDL) code used to program FPGAs, which often require vendor specific "primitives" to use the full feature set of the device.

An attempt to provide some measure of interoperability is the use of intermediate representations, as used by popular compiler frameworks such as LLVM. Beyond multicore CPUs, in many cases, the C programming language, or a subset thereof, is treated as a "portable" assembly language that can be compiled by platform-specific compilers without too much modification. Examples of this trend include NVIDIA's CUDA framework for GPUs or Xilinx's Vivado HLS tools for FPGAs.

Even so, there are three problems with the platform-specific approach that prevent it from being interoperable, even if a relatively portable language such as C is used:

1. Inconsistent Feature Support

The subset of operations supported is determined by the vendor, so beyond the simplest of arithmetic and memory operations, there is no guarantee a required function or library would be supported.

An example of this is support for dynamic memory allocation. In multicore CPU programming, code is often optimised by allocating and deallocating working memory resources as needed. However, in many platforms, such as GPU and FPGA programming frameworks, all memory declarations have to be made at compile time, making the memory use determined by the worst possible case.

2. Varied Compilation Interfaces

The *flow* from source code to implementation upon the target platform differs significantly between vendors, requiring inconsistent degrees of user intervention.

For example, both NVIDIA's CUDA and Vivado HLS accept a similar subset of ANSI C for execution on the devices both vendors provide [12, 13].

CUDA requires code to be expressed within a task parallel framework, with the C functions being executed on the GPU identified explicitly by the programmer as *kernels*. Inside these kernels, special variables are made available to the programmer to distinguish between different kernel instances or *threads*, and hence code has to be refactored to make use of these special variables.

A Xilinx implementation of the same code would look considerably different, as the programmer would identify the function(s) that should be implemented in the reconfigurable fabric, and also specify the nature of interfaces used to communicate with these function. Once a design had been synthesised from the source code, the designer would still need to implement it within a system architecture that cannot currently be expressed in C.

3. Inconsistent Optimisation Approaches

Optimisation is another area where there is less standardisation. Most vendors require the use of specific source code annotations or libraries which do not translate between architectures.

To use the examples of CUDA and Vivado HLS again, task parallelism is expressed quite differently. For CUDA, the programmer is required to specify the number of threads when calling the kernel in their code. In Vivado HLS, this is done implicitly within the code being synthesised; by calling the same function multiple times at the same scope level.

2.1.2. The Rise of Heterogeneous Computing Standards

The need for interoperability between different computing architectures has prompted the promulgation of heterogeneous computing standards in recent years, such as the Open Compute Language (OpenCL) [14], Open Spatial Language (OpenSPL) [15] and Open Accelerators (OpenAcc) [16] standards. Similar to earlier heterogeneous standards, such as POSIX (IEEE 1003) for CPUs, and Verilog (IEEE 1364), VHDL (IEEE 1076) and SystemC (IEEE 1666) for FPGAs and ASICs, these standards represent an agreement between vendors to support a carefully defined Application Program Interface (API)[1].

The newer heterogeneous computing standards are at a higher level of abstraction, and cover a broader set of platforms, including multicore CPUs, GPUs and FPGAs. These modern standards also address the three issues outlined in the Platform-specific Approaches subsection: a core set of features is universally supported, standard compilation and run-time APIs are defined and a set of generic optimisations are also specified.

To examine this trend I describe and comment upon two recent, distinct examples: OpenCL, a standard which has been widely adopted and established over the past few years, and OpenSPL, a more recent effort from a single vendor.

Both OpenCL and OpenSPL assume a host-accelerator system organisation model, with programs or kernels run upon the accelerator device that interfaces with code on the host, which is a conventional CPU-based platform. However, while OpenCL kernels are imperative in nature, OpenSPL is organised around dataflow principles, and so represents a dramatic departure from conventional, CPU-based programming languages.

In both cases I use the kernel example given as C code function in Listing 2.1, of a scaled vector sum of floating point values, such as may be found in level 1 of the popular Basic Linear Algebra Subroutine (BLAS) library.

[1]And apparently to use the word “Open” in their name

Listing 2.1: Scaled vector sum example code

```
void vector_sum(
    const int N,
    const float alpha,
    float *x,
    const float beta,
    float *y,
    float *z)
{
  for(int i=0; i<N; ++i)
    z[i] = alpha * x[i] + beta * y[i];
}
```

Software	Hardware	Memory
Context	Device	Global
Work-group	Compute Unit	Local
Work-item	Processing Element	Private

Table 2.1.: OpenCL Abstractions. The rows indicate the minimum scope of access for that level of abstraction

OpenCL

OpenCL is a task parallel standard that is organised around a *host*, a conventional CPU, that interfaces with one or more heterogeneous accelerator *devices*, executing *kernels* [14]. The devices are composed of one or more *Compute Units*, that are in turn composed of one or more *processing elements*. The devices execute many, out-of-order kernel instances or *work-items* upon its processing elements. Work-items are organised into *work-groups*, which will always be processed within the same compute unit. The memory hierarchy is explicit, requiring the programmer to qualify whether variables will be stored in *global, local* or *private* memory, which are typically implemented in memories of decreasing size and access latency. Table 2.1 provides an overview of these abstractions.

The code used to write OpenCL kernels, of which the scaled vector sum example is given in Listing 2.2, is a subset of ANSI C (IEEE 1003.1-1988), most notably without support for dynamic memory allocation. The host interface is a C API that interfaces with an OpenCL run-time driver on the host system, which is provided by the vendor of the targeted hardware. Bindings for the host API exist in many higher level programming languages such as C++ and Python. Through its host API, the standard also offers the means to query the computing resources and memory available both prior to, and during execution.

In terms of optimisation, in addition to task parallelism being made explicit, OpenCL allows for vendor-provided or "native" implementations of certain mathematical functions, such as *sin* or *cos*. The native functions will be more efficient, however will deviate from the IEEE 754 floating point standard. The OpenCL standard also also allows for more complex custom functionality through vendor extensions to the core standard.

Listing 2.2: OpenCL scaled vector sum kernel

```
kernel void vector_sum(
    const float alpha,
    global float *x,
    const float beta,
    global float *y,
    global float *z)
{
  //getting unique ID
  int i = get_global_id(0);

  //Perfoming the vector computation
  z[i] = alpha * x[i] + beta * y[i];
}
```

Driven by the pressing need for software development infrastructure for heterogeneous computing, many influential vendors such as Intel, AMD, ARM, NVIDIA, Xilinx and Altera have joined the Khronos consortium that manages the standard. More importantly, many of these vendors provide the Software Development Kits (SDKs) and run-time support required to execute OpenCL code on their hardware. The standard draws inspiration from the success of NVIDIA's CUDA framework for GPU computing through its use of easy-to-use abstractions and broad API. As a result of its wide support, choosing to use OpenCL will not necessarily dictate the vendor or even the architecture that the code will eventually run on.

A significant challenge in the use of OpenCL is the need for programmers to manage memory locality. For programmers unfamiliar with memory hierarchies, and techniques for using them effectively, this is quite a daunting prospect. On the other end of the spectrum, the standard is a victim of its own success, as it guarantees equivalence of computational result but makes no guarantees as to the portability of performance. A programmer might be tempted to simply reuse the same kernel code between radically different architectures, and not do the necessary code refactoring that might be required to extract that platform's best performance.

OpenSPL

The OpenSPL standard conceptualises computing systems as being organised spatially as opposed to temporally, as is the case in imperative programming [15]. To enable this spatial paradigm, OpenSPL uses a dataflow approach to computing. The OpenSPL consortium currently has only one hardware vendor, Maxeler, and a small number of members.

OpenSPL allows the programmer to define the interaction of multiple, simultaneously running *kernels* via flows of data, implemented on a *spatial computing substrate*. Three types of memory are defined: single-value *scalars*, as well as *Fast* and *Slow Memories*, with the expectation being that size will be correlated with memory access latency. The standard is aimed at enabling domain experts to generate optimal computational structures for particular applications. An example of an OpenSPL kernel is given in Listing 2.3.

Listing 2.3: OpenSPL scaled vector sum kernel, as it would be defined within Maxeler tools

```
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class MaxelerVectorSumKernel extends Kernel
{
    protected MaxelerVectorSumKernel(KernelParameters parameters)
  {
    super(parameters);

    DFEVar alpha = io.scalarInput("alpha",dfeFloat(8,24));
    DFEVar beta = io.scalarInput("beta",dfeFloat(8,24));

    DFEVar a = io.input("a", dfeFloat(8,24));
    DFEVar b = io.input("b", dfeFloat(8,24));

    io.output("output", a * alpha + b * beta, dfeFloat(8,24));
  }
}
```

Unlike OpenCL, OpenSPL does not proscribe a host-accelerator system organisation nor a host API, however in practice this is the configuration used by the Maxeler, the only vendor that currently supports the standard. The tools provided by Maxeler are however capable of targeting FPGA-based platforms from two of the major vendors, Xilinx and Altera.

OpenSPL will only see adoption in those instances where the benefits from using it offsets the cost of refactoring code and algorithms to fit within the paradigm. This is also in contrast to OpenCL, which allows for the relatively easy porting of legacy C code. Furthermore, given the small size of the supporting consortium relative to OpenCL, it will likely struggle to gain the same widespread acceptance.

2.1.3. Heterogeneous Computing Frameworks

Beyond standards for heterogeneous computing and implementations of these standards, general purpose heterogeneous programming frameworks are seeing considerable attention from academia, with projects such as Qilin [17], Harmonic [18], LegUP [19] and industry, in the form of Liquid Metal [1] and Exochi [20]. These frameworks seek to abstract away many of the platform specific details of the components that make up the programmer's heterogeneous computing systems, often presenting a uniform interface, while accessing and using multiple backend run-time environments automatically.

However, in addressing the interoperability challenge, these approaches potentially obscure the characteristics of the underlying architectures. Generally these approaches move the conceptual burden of working with a multiple heterogeneous systems with distinct, observable characteristics onto an abstract level defined by the framework, and hence not easily observable by the programmer.

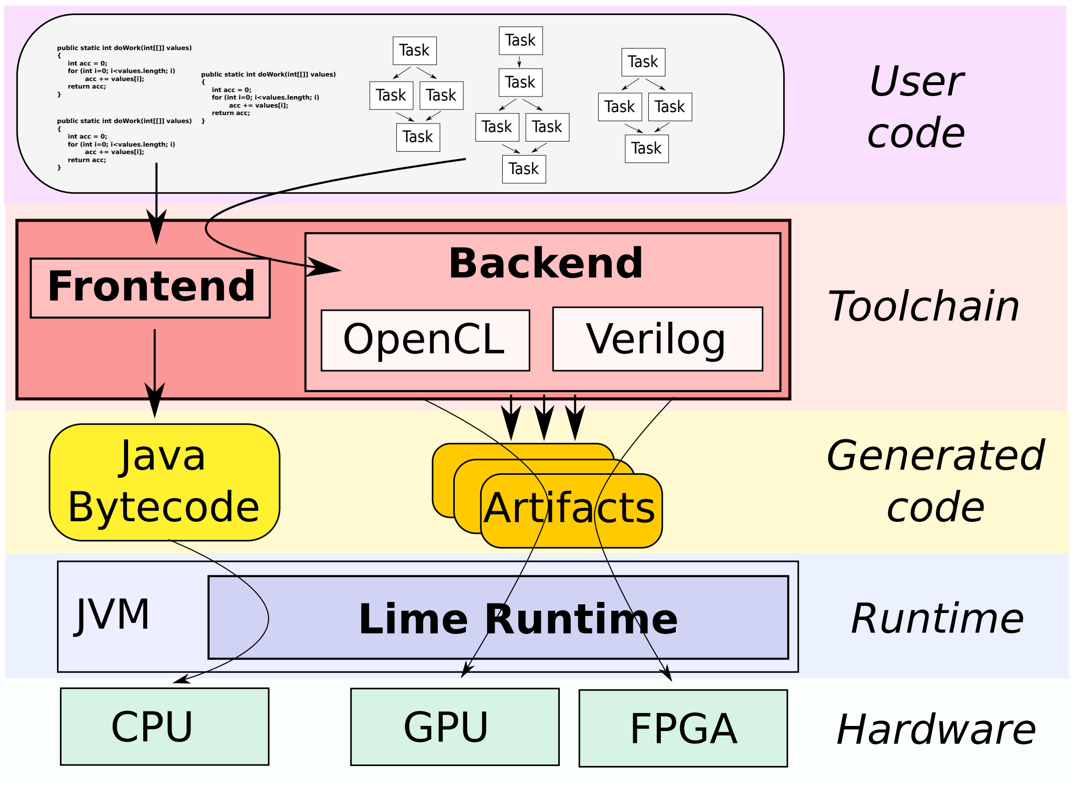


Figure 2.1.: Overview of Liquid Metal System [1]

The potential danger of a high level of abstraction have been demonstrated experimentally, using a popular parallel programming framework, MapReduce [21], as a case study. In a study by Ahmad et al. [11] it has been shown that while MapReduce intuitively seems to support heterogeneous parallel execution, however if used naively, it under performs when using two CPU architectures of differing capabilities. The scale of the under performance is to such a degree that only using a homogeneous subset of the available resources would achieve the same results more quickly.

This argument and cautionary case makes it clear that the characteristics of both the application and the heterogeneous platforms concerned need to be taken into account in order to realise the potential of heterogeneous computing devices. The remainder of this section is a brief survey of several general heterogeneous computing frameworks:

Liquid Metal

Liquid Metal is a compiler and run-time system for LIME, an Object Orientated Programming (OOP) language for programming heterogeneous systems, developed at IBM [1, 22]. The goal of the framework is to make the benefits of heterogeneous computing more accessible to a wider audience, particularly those benefits realised through reconfigurable hardware. In order to achieve this goal, the framework is able to compile a single OOP language, LIME, for multicore CPUs, GPUs and FPGAs, and support parallel execution upon systems composed of multiple heterogeneous platforms. A diagrammatic overview of the Liquid Metal system is given in Figure 2.1.

The LIME language is based upon Java, and can be integrated with existing Java code. The language introduces various features so as to allow the programmer to make task, data and pipeline parallelism explicit. Notably task and pipeline parallelism is enabled by the introduction of dataflow or stream computing concepts, in a similar manner to how event semantics are introduced in SystemC. Units of code are defined as *Tasks* and *connected* together by the programmer, allowing for the creation of *task-graphs* in a dataflow manner, similar to what is envisioned in the OpenSPL standard.

The focus of the Liquid Metal work is in providing a complete modern programming language definition and heterogeneous run-time support system. The range of platforms supported is broad relative to other frameworks, including multicore CPUs, GPUs (using an OpenCL backend) and FPGAs (using a Verilog backend). Performance figures [22] over native Java byte code show a *4.8x* and *32.5x* improvement for multicore CPU implementations, and *12x* to *430x* improvement for GPUs for a set of parallel computing benchmarks. When compared to OpenCL code written by a programmer running upon the same platforms, the Liquid Metal implementations' performance was between 75% and 140% of the programmer code. These results suggest that the framework is able to produce implementations from a high level of abstraction that are able to take advantage of the parallel compute resources almost as well as programmer written code.

A potential weakness of the framework is the abstraction model used. The level of abstraction, while higher than low level OpenCL code, is still different from conventional Java in the same way that SystemC is different from C++. Furthermore, by integrating the dataflow semantics into the Java language, it is possible that programmers will become confused, and default back to familiar Java constructs.

Another weakness is that while supporting execution on multiple heterogeneous platforms in parallel, the Liquid Metal run-time does not automatically partition and schedule work across the available platforms so as to balance communication and computation, or according to other user goals, such as energy optimisation. This requires the programmer to identify and map sections of the application to appropriate platforms themselves, a task requiring considerable insight into the nature of the platforms available, as well as the nature of the Liquid Metal implementations being produced. It should be noted though that the developers of Liquid Metal view this functionality as orthogonal to their work, as an area which can be explored using the framework as a compiler and run-time system.

Exochi

Exochi, an early effort from Intel, is both a system architecture that represents tightly-coupled heterogeneous computing platforms, such as on-chip GPUs, as ISA-based architectural resources (EXO), as well as a C++ programming environment that allows for accelerator-specific code (CHI) [20]. The modified C++ compiler extends OpenMP pragmas to expose these heterogeneous resources to the programmer.

An Exochi prototype system comprised of a multicore CPU and GPU achieved performance speedups of up to *12x* relative to the multicore CPU for a set of image and video processing tasks, suggesting that the heterogeneous resources of the system were being harnessed productively.

However these speedups were only achieved through an ideal workload partitioning between the CPU and GPU, found by running the tasks on both platforms independently and partitioning accordingly. Performance figures relative to programmer GPU code were not reported, however, due to its bypassing of operating system mechanisms for accessing heterogeneous resources, there is potential for improvement over programmer code.

The framework maintains the look and feel of current approaches towards high performance computing through the use of C++ and OpenMP pragmas, however the wideness of its use is limited by the low level, Intel-specific architectural features used.

Also, similar to Liquid Metal, this work did not go beyond providing support for heterogeneous execution - it is up to the programmer to select how work should be partitioned between the available processing platforms.

Qilin

Qilin is an academic experimental application framework for CPU and GPU computing systems [17]. In addition to providing a uniform API for the parallel execution of linear algebra domain functions upon both multicore CPUs and GPUs, the framework is able to automatically partition work between the available computing platforms, a feature generally lacking in general purpose heterogeneous computing frameworks.

The *adaptive mapping* introduced by the framework uses previous runs of particular task and parameter combinations to predict the relative run-times upon the available CPUs and GPUs using linear models. The linear models for the available platforms are then used to create a system of equations, which are then solved to find a partitioning of work that minimises the run-time.

Using an ideal partitioning of work between GPU and CPU, the mean latency of the framework is *9.9x* faster than serial code for a wide set of benchmarks. The adaptive mapping achieves a *9.3x* improvement over the serial without programmer intervention, close to the ideal case. The adaptive mapping also outperforms the single platform implementations in all cases, thus only making use of the heterogeneous resources to a useful degree, avoiding the trap identified by Ahmad et al [11].

The work is however limited in considering only a single CPU and GPU processor combination. Furthermore, the framework only supports tasks which are capable of being partitioned, limiting itself to linear algebra functions, arguably making it a domain specific application framework, similar to those described in Section 2.2.3.

2.2. Domain Specific Computing

Domain specific computing is the study of computing organised around groupings of applications, called domains. Although Domain Specific Languages (DSLs), programming languages that cater to certain domains, have been in use for over 40 years, academic consideration of the topic in its own right has only really occurred in the past 20 years [2, 3, 23, 24, 25]. While there are many motivations for domain specific computing, by far the most powerful is the ease of access it affords to programmers with little to no formal training. Domain specific approaches allow domain end users to clearly express their intent, i.e. to program, at a high level of abstraction, with concepts and even jargon that they are familiar with. This ease of expression results in significant productivity benefits [2, 26].

The chief limitation of a domain specific approach is the focus upon a single application domain. If what the domain programmer requires is outside the confines of the domain, as defined by the system programmer that implemented the domain specific tool, there is no guarantee that the functionality is supported.

A further limitation is the upfront development effort and cost of defining the application domain, and providing support for its execution. This can account for many person-years of development time before even the first useful program is run. Indeed, Fowler and Parson [2] and Mernik et al [3] suggest that the first and one of the most important activities in domain specific system development is justifying its necessity in the first case.

In this subsection, I survey the literature of Domain Specific Programming, and its relevance to supporting the use of heterogeneous computing. I begin by considering the different classes of users within domain specific programming. By considering the usage modes of stakeholders, and their spectrum of computing knowledge, I show the necessity of the formulation of domain specific abstractions. I then describe the process of developing of domain specific computing systems: from identifying the domain-knowledge informed semantic model, to designing the means of expression and finally the implementation of the domain program. Finally, I consider a relatively new development, that of domain specific heterogeneous computing frameworks, and how domain specificity is advantageous in the heterogeneous computing context.

2.2.1. Domain Specific Programming Stakeholders

The literature concerned with the study of programming by end users, or end user programming, is based upon an empirical or ethnographic approach to the use of computing as opposed to more formal methods [27, 28, 29]. This ethnographic approach entails studying how these users make use of software in practice through reviewing code produced by, and interviewing, with these end users. End user programming is considered to be systems or environments which *'allow users to create useful applications with only a few hours of instruction'* [28]. This is almost always achieved through the use of domain specific abstractions, allowing the end user to describe their intent using concepts that they are already familiar with. Hence, I refer to these types of users as domain programmers.

Studies from this field have found that in large, single sector organisations in public or corporate settings, where domain specific systems are commonly used, a spectrum of computing users exist [28]:

- **The system programmer** that is highly knowledgeable and skilled in computing, and has varying degrees of knowledge of the application domain. They are also confident and supported in their computing work by the organisation's management.

- **The local developer**[2] that is skilled and confident in their use of computing, as well as having a high degree of knowledge about the application domain. While not necessarily officially recognised by the organisation management, these role-players are locally identified as experts by their colleagues, effectively bridging the gap between the application domain and the computing resources.

- **The domain programmer**[3] that is highly knowledgeable of the application domain, but largely unskilled or ignorant of computing. Both Nardi [28] and Blackwell et al [27] suggest that there are ten times as many domain or end user programmers as formally trained programmers.

This stratification of users has implications for the development of domain specific computing systems. The asymmetries of computing and domain knowledge suggest that beyond a more natural way to express computational tasks, domain programmers require a means to reason about their computations in terms of concepts they are familiar with. However the system programmers need an unambiguous specification to implement. The application domain represents a compromise between the two groups, restricting the domain programmer to what the system programmer can practically implement, however doing so in such a way that provides useful abstractions. The local developer is often critical in realising this vision, as they can aide in mediating the shortcomings of system programmers, whilst helping clarifying the domain programmer's needs.

A second significant finding is that within a particular application area, such as Computer Aided Design, there are typically ten to fifteen high level functions that are disproportionately used by end users [28]. Such a power law distribution is useful to consider when supporting an application domain, as it suggests implementing these heavily used functions efficiently will address the needs of a significant number of end users.

2.2.2. Domain Specific System Development Process

The domain specific system development process in Figure 2.2 is a synthesis of those suggested by Fowler and Parsons [2] and Mernik et al [3]. The process is comprised of three distinct phases, Analysis, Design and Implementation, that are described below. However, similar to software development, developing a domain specific computing system is often iterative, with the process being iterated over several times.

Analysis

The purpose of the analysis phase is to analyse the application domain, and create a semantic model[4] that captures all of the concepts and behaviours within the targeted application domain [2]. The semantic model provides the vocabulary for the eventual domain specific means of expression, whether a language or framework. The semantic model represents a *conceptual*

[2]Various alternative, more colourful titles for this role have been recorded: "Tinkerer", "Translator", "Gardeners"

[3]Often called an end-user programmer

[4]Fowler and Parsons [2] distinguish between a semantic and domain model, describing the latter as behaviourally richer description.

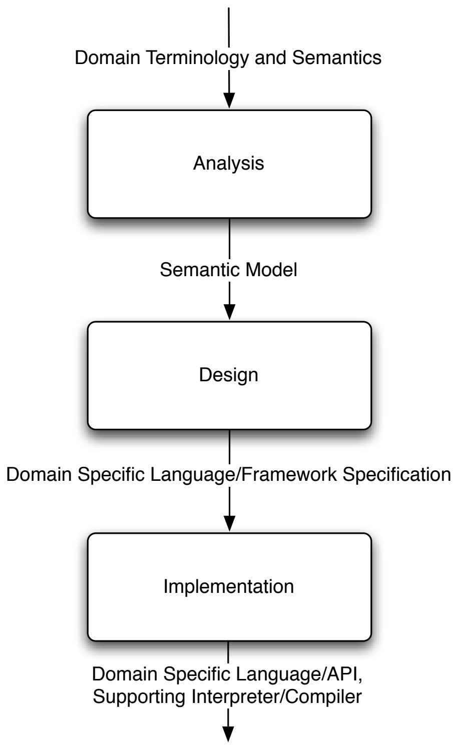

Domain Terminology and Semantics
Analysis
Semantic Model
Design
Domain Specific Language/Framework Specification
Implementation
Domain Specific Language/API,
Supporting Interpreter/Compiler

Figure 2.2.: Domain Specific System Development Process [2, 3]

interface, the means by which the domain user can think and reason about their computations, that can still be captured unambiguously for execution. Fowler and Parsons [2] use the example of a state machine domain, suggesting that the semantic model would be an object model with classes for states and events.

The semantic model is formed based upon analysis of domain concepts and jargon, as well as consultation with domain experts. For a given domain there should only be one semantic model, while there might be many domain specific means of expression that work with this model.

Design

In the design phase, a means of expression for describing computation in accordance with the semantic model is created [2, 3]. This domain specific means of expression should give the domain programmer the ability to manipulate the domain abstractions provided. Hence the domain programmer, through the domain specific means of expression, is describing or configuring an aspect of a system, rather than describing a complete system[5].

The key design decision that has to be be made by the system programmer, the stakeholder described in Section 2.2.1, is whether the means of expression for the user should be a Domain Specific Language (DSL) versus an application framework. DSLs are often, but not necessarily, Turing complete programming languages built around the application domain's semantic model. An application framework is a set of classes or library [23, 30], which is implemented in a 3rd generation programming language, such as Java or C++, that makes available an API for implementing instances from the domain's semantic model.

[5]For this reason, Fowler and Parsons [2] argue that many of the configuration files within UNIX-based operating systems are domain specific in nature.

Listing 2.4: Random scaled vector sum in MATLAB, a linear algebra, external domain specific language

```
alpha = rand();
beta = rand();
x = rand(1000,1);
y = rand(1000,1);

z = x * alpha + y * beta;
```

Listing 2.5: Random scaled vector sum using the Numpy library in Python as a linear algebra application framework

```
import numpy,numpy.random
alpha = numpy.random.random()
beta = numpy.random.random()
x = numpy.random.random(1000)
y = numpy.random.random(1000)

z = numpy.sum(numpy.multiply(x, alpha), numpy.multply(y, beta))
```

Both DSLs and application frameworks limit the computations that may be expressed to a particular domain, however the key difference between the two is the degree of fluency that a standalone language versus a library enables. Whilst this fluency might make the DSL more easily understood by the domain users, there is additional implementation effort required to support it. Somewhere between the two are internal DSLs, which are implemented within a host, general purpose language, but make use of techniques such as syntactical operator overloading and method chaining to achieve some of the fluency of a standalone language. Hence, completely standalone DSLs are referred to as external [2] or formal DSLs [3] to distinguish from these internal DSLs.

An illustration of the distinction between application frameworks, internal and external DSLs is provided in Listings 2.4, 2.5 and 2.6. The Numpy Python application framework by comparison to the MATLAB external DSL is considerably less easy to read, but when Numpy is used as an internal DSL, through pre-emptive library loading and the overloading of arithmetic operators, a similar fluency to that seen in the external Matlab DSL is achieved.

Listing 2.6: Random scaled vector sum using Numpy as an internal domain specific language

```
alpha = random()
beta = random()
x = random(1000)
y = random(1000)

z = x * alpha + y * beta
```

The outcome of the design phase is a specification for the DSL or API for the application framework required to support the semantic model of the desired application domain.

Implementation

The implementation phase takes the domain specific DSL or API specification, and develops the necessary infrastructure for supporting the execution of programs described in accordance with the specification.

If an external DSL has been designed, then the supporting infrastructure must be capable of parsing programs written with it, and then implementing the computations described. This is essentially identical to implementing a compiler for the DSL, although the DSL compiler will conceivably be easier to implement due to the limited number of domain data types and functions that need to be supported.

In the case of application frameworks and internal DSLs, usually the host language compilation and execution infrastructure may be used, hence dramatically reducing the implementation effort. An example of a linear algebra application framework is LAPACK [31], which is actually in turn built upon BLAS [32], a lower level, linear algebra domain library. However if an internal DSL has been created, usually some degree of transformation or processing often has to occur upon the specified program prior to direct execution.

A popular implementation technique [2] is to generate code for another, general purpose programming language, such as C, from the domain specific task description, and then to compile the resulting code using compilers of the generated code's language. An example of this approach is SPIRAL [33], a framework for generating efficient Digital Signal Processing (DSP) code. The language of the generated code is often at a lower level of abstraction than the domain specific description. While requiring additional implementation effort than the direct execution of internal DSLs and application frameworks, code generation allows for the execution environment to be separated from the environment which the domain specific task is described and compiled in. So, even if a high level, sophisticated compiler framework is used for the domain specific task, a relatively simple target can be used to execute the resulting implementation from generated code.

The output from the implementation phase should be the means to execute programs which conform to the DSL or application framework's specification. What form this means takes depends upon the nature of the application domain. It might be a software system, such as an operating system service, that is capable of parsing a domain specific configuration so as to control its behaviour, or it might be a compilation and run-time framework for transforming domain specific task descriptions into standalone executables.

2.2.3. Domain Specific Heterogeneous Computing

While in the previous two subsections, I have looked at the concept of domain specific computing with the assumed context of conventional, x86 computing. In this subsection I consider it when applied to heterogeneous computing. There has been work by Brown et al [4] illustrating how domain-specific methods may be used to make heterogeneous computing systems more accessible.

Implementation	**Productivity Improvement** $(\frac{\mathbf{LoC_{Gen}}}{\mathbf{LoC_{DS}}})$	**Performance Improvement** $(\frac{\mathbf{Latency_{Gen}}}{\mathbf{Latency_{DS}}})$
Contessa [34]	4.65x	1.94x
Green-Marl [36]	5.14x	5.43x
Delite [4]	1.28x	6.24x

Table 2.2.: Productivity and performance benefits of domain specific heterogeneous approaches over general approaches. Productivity metric is Lines of Code (LoC), performance metric is wall-time latency.

By providing a single means of expression and a unified run-time environment, the complexities of interacting with different heterogeneous platforms can be obscured from the end user.

However, another important finding in recent years is that domain specific abstractions enable improved performance over general approaches in the heterogeneous computing context [4, 34, 35, 36], as evidenced by three implementations detailed in Table 2.2. All three bodies of work suggest improved performance is achieved through the limitations imposed by the domain. By only having to focus on a limited subset of operations and dependency relationships, it is easier for system to automatically extracting program features such as task and data parallelism, and hence can provide an automatic, yet efficient execution on a range of platforms.

The remainder of this section describes these three domain specific, heterogeneous computing implementations:

Contessa

Contessa [34] is an external DSL for describing path-based Monte Carlo simulations that generates C++ for multicore CPUs and uses the Catapult high level synthesis tools from Calypto to target FPGAs. The aim of Contessa is to allow for the high level, platform independent description of Monte Carlo simulation-based applications, but still achieve good performance upon the platforms targeted. Contessa not only makes the inherent task and pipeline parallelism in Monte Carlo simulations explicit, but also enforces a stricter functional programming approach without overly inhibiting programmers. As a result, efficient FPGA and multicore CPU implementations can be generated from a Contessa description without programmer intervention.

Contessa is however limited in supporting only multicore CPUs and FPGAs, and is really only focused on FPGAs. There is also no support for partitioning work between CPU and FPGA resources, manually or automatically.

Green-Marl

Green-Marl [36] is an external DSL for graph analysis algorithms. The language attempts to expose as much data-level parallelism as possible in the algorithms, through language constructs for parallel graph operations as well as speculatively processing operations in parallel until conflicts are detected. Additional architecture independent and dependent optimisations are used to generate an efficient implementation of the algorithm under consideration in C++.

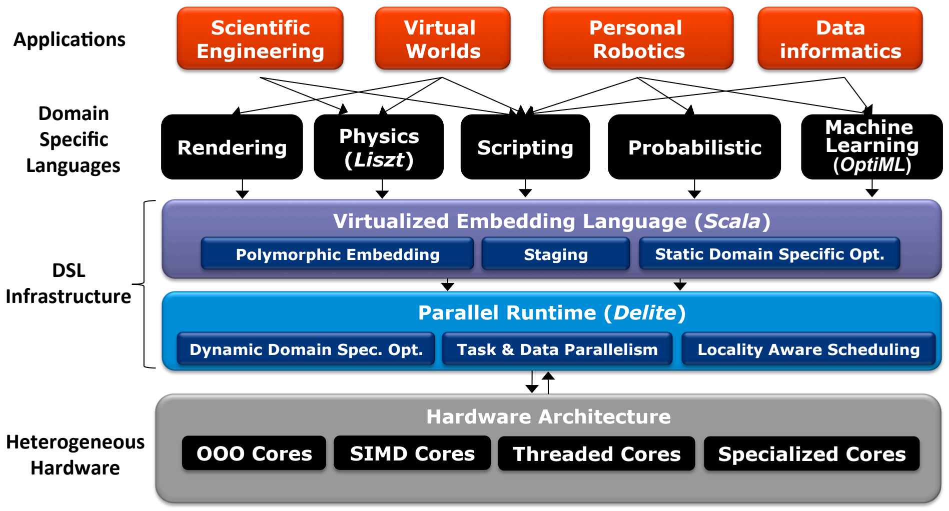


Figure 2.3.: Overview of Delite System Architecture [4]

This work is of particular interest as graph theory is itself a common programming abstraction, upon which a diverse range of applications such as data analytics, social network analysis and bioinformatics may be mapped. The chief limitations of the work is that only multicore CPUs are supported. It is also assumed that only one multicore CPU is being utilised, and hence there is no support for partitioning of work.

Delite

Delite is different from the other two bodies of work, in that it is a framework that advocates the use of language “virtualisation” in order to meet the productivity and performance requirements of programmers, whilst making optimal use of heterogeneous computing resources [4, 37]. Delite is both a framework for creating implicitly parallel internal DSLs as well as a dynamic run-time for running applications created using such languages, as illustrated in Figure 2.3.

The strength of Delite framework is in the breadth of the experimental work undertaken in three domains, Machine Learning, Data Querying and Graph Analysis. In all three cases the framework has been able to deliver consistently better results than competing approaches. However, the implementations have thus far has been confined to one multicore system with the use of only one accelerator, a NVIDIA GPU. The run-time system does attempt to partition and schedule work so as to maximise throughput, however it is only able to do so for sections of the code that are amenable to static analysis.

2.3. Distributed Computing

The problem of distributing computational tasks to heterogeneous computing resources has been widely studied for almost 40 years [38, 39, 40, 41, 42, 43, 44, 45, 46]. Starting with computing grids located in specialist facilities to ad-hoc clusters created for the duration of hours using IaaS computing infrastructure, the question of how to relate applications and resources with a view towards efficient execution has proved remarkably resilient to definitive solution.

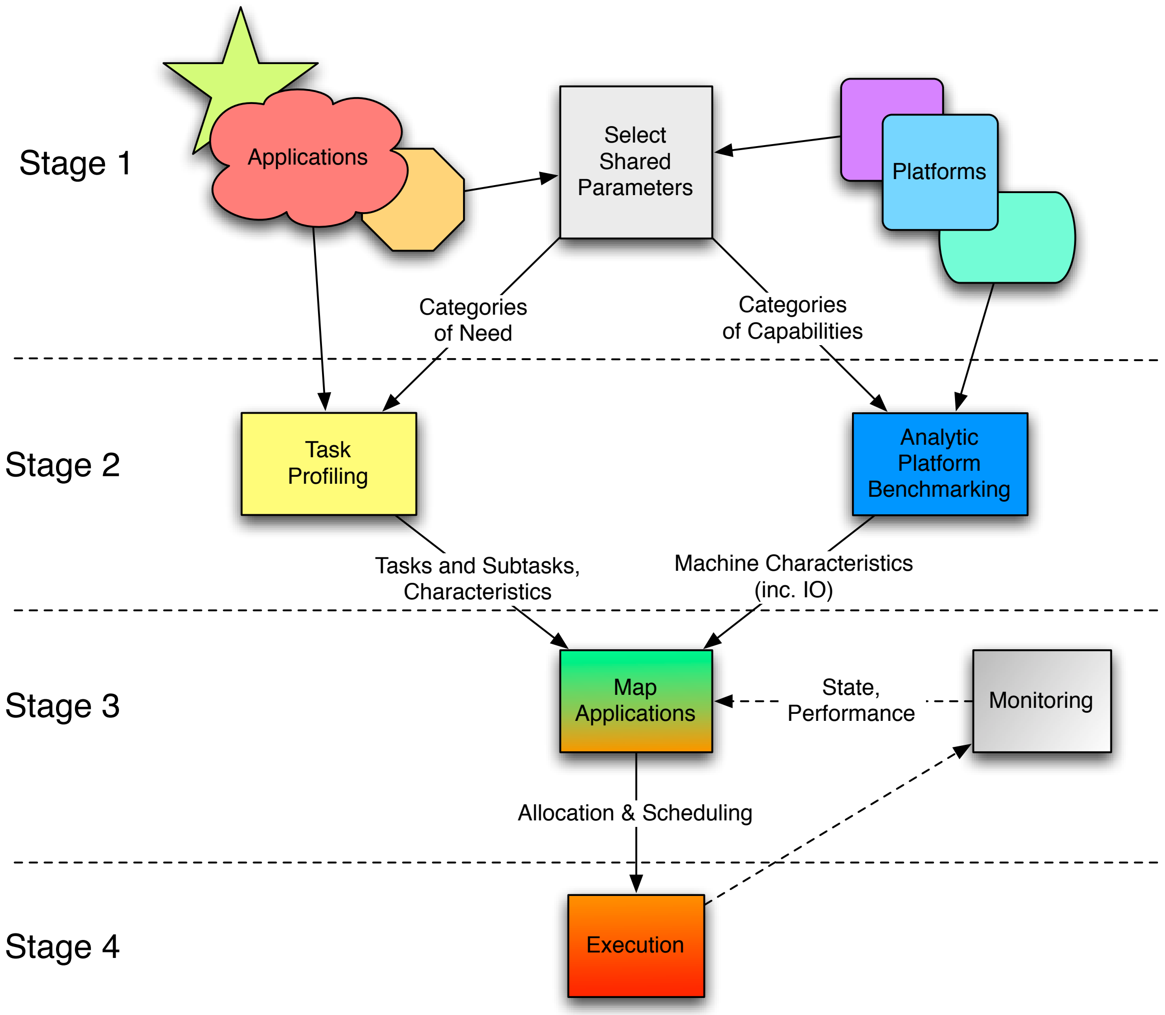


Figure 2.4.: Model for automating support of heterogeneous, distributed systems [38]

Braun [38] formulated a four stage process for automating support for distributed, heterogeneous computing, as provided in Figure 2.4. In this subsection, I consider different approaches for Stages 2 and 3 within this process. Firstly, I consider how tasks and platforms should be characterised so as to enable mapping. I then consider the challenge posed by mapping and different approaches to addressing it.

2.3.1. Task and Platform Characterisation

Characterising the execution of tasks upon heterogeneous computing platform is comprised of three interrelated activities:

1. **Task Profiling:** identifies the atomic (i.e. indivisible) tasks that comprises the current application. These tasks can then be further qualified by performing analysis or profiling of the task code. A key insight from Khokhar et al [39] is that profiling should determine the parallel execution modes possible for the given task. An increasingly popular approach is to get the programmer to make the parallel execution modes for tasks explicit, either through a specially designed API [17] or by embedding this within the language itself [4].

2. **Analytic Platform Benchmarking:** identifies the capabilities of the heterogeneous computational platforms available. Another insight from Khockar et al [39] is that this process mirrors what is occurring in the task profiling activity, i.e. detailing how well the platform supports different parallel execution modes. A heterogeneous benchmark such as Rodinia [47] could be used for this purpose, or a representative subset of the current tasks, as used in Qilin [17].

3. **Task-Platform Characterisation:** synthesises the data from the two previous activities, which results in models of how the specified tasks will execute upon the available resources. Grewe's work [44] illustrates a sophisticated machine learning-based approach for doing so. A platform modelling approach [48, 49] could be used to model the target computing resources, and evaluate how the profiled tasks would execute upon it. Alternatively, Ardagna et al [50] show how measurements of the performance of the application itself can be used to characterise the platform directly.

The characterisation activity is often not distinguished from partitioning of tasks upon the platforms [17, 44], although there have been notable exceptions such as Kraft et al [51]. I believe that maintaining this separation is useful, as demonstrated by the Roofline model [52], as it allows for the quality of the characterisation activities to be evaluated independently from the mapping approach that is being used. The experimentation undertaken with MODAClouds [50] further illustrates how a modelling approach can be distinct from the mapping process.

2.3.2. The Mapping Problem

When considering the allocation of tasks to heterogeneous computing resources, the general scenario considered in the literature, i.e. [40, 42, 44, 45, 46, 41, 53], is a set of τ independent or atomic tasks being partitioned across ρ heterogeneous platforms. It is assumed that a task will occupy any of the computing resource completely if allocated to that resource for a period of time known prior to execution. It is also commonly assumed that the partitioning is being performed statically, in advance of the execution of any of the tasks. The objective is to minimise the makespan, which is a single scaler value in this scenario.

The makespan is the latency (L) from when the first task is initiated until the last result returned for the task set. As the tasks are being evaluated on multiple platforms, the makespan is equivalent to the longest time it takes for any of the platforms to return the results of the tasks allocated to it. In this context, the makespan is given by taking the maximum of platform latencies, which is given by the sum of the tasks on each platform. The latency of each task upon each platform is found by taking the element-wise or Hadamard product of the task allocation ($\boldsymbol{A}$) and task latency matrix ($\mathbf{X}$). Hence the makespan can be expressed as a function of $\boldsymbol{A}$ and $\mathbf{X}$, i.e. $F_L(\boldsymbol{A}, \boldsymbol{X})$, as defined below.

Minimising the makespan for tasks upon platforms with *a priori* knowledge or predictions of the execution time of atomic tasks ($\mathbf{X}$) is a well studied problem. As I show in (2.1), this problem can be expressed formally as a 0-1 or binary integer linear programming (ILP) problem. Karp [54] famously demonstrated that binary ILP problems have NP-complete complexity.

$$\begin{aligned} \underset{\boldsymbol{A}\in\{0,1\}^{\mu\times\tau}}{\text{minimise}} \quad & F_L(\boldsymbol{A},\boldsymbol{X}) \quad \boldsymbol{X}\in\mathbb{R}_+^{\mu\times\tau} \\ \text{subject to} \quad & \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1,2,\ldots,\tau \end{aligned} \tag{2.1}$$

where:

$$F_L(\boldsymbol{A},\boldsymbol{X}) = \max((\boldsymbol{A}\circ\boldsymbol{X})\cdot\mathbf{1})$$

However, in the above formulation, similar to the literature [40], only latency performance is considered. Whilst latency is important, throughput, cost and computational quality measures are orthogonal to it, and hence any mapping approach should provide programmers with the means to balance these objectives.

Equation 2.2 includes these additional considerations as further constraints (T, C and Q for throughput, cost and quality respectively) that have to be satisfied.

$$\begin{aligned} \underset{\boldsymbol{A}\in\{0,1\}^{\mu\times\tau}}{\text{optimise}} \quad & F_L(\boldsymbol{A}) \\ \text{subject to} \quad & \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1,2,\ldots,\tau \\ & F_T(\boldsymbol{A}) = T \quad T\in\mathbb{R}_+ \\ & F_C(\boldsymbol{A}) = C \quad C\in\mathbb{R}_+ \\ & F_Q(\boldsymbol{A}) = Q \quad Q\in\mathbb{R}_+ \end{aligned} \tag{2.2}$$

2.3.3. Mapping Approaches

Surveying the literature, there are three suggested approaches to the mapping problem:

- Naive Heuristics [40, 42, 46, 55, 56]: a simple algorithmic rule is applied to allocate tasks to the available resources. Under specified circumstances such a rule might achieve a provably optimal allocation of tasks, and there is usually a worst case bound on the quality of the solution relative to the optimal solution. Some heuristics require estimates of task runtime in advance, whilst some do not [40].

- Machine Learning [44, 45, 57, 58]: a feasible task-platform allocation is improved using global optimisation techniques such as the unconstrained simplex algorithm, simulated annealing or genetic algorithms. As the optimisation problem is convex, at the worst these techniques can confirm the quality of the starting solution. Such approaches require estimates of task run-time to compute the objective function.

- Integer Linear Programming [41, 53, 59]: the optimisation problem formulated above can be solved using ILP techniques, which in addition to applying the global optimisation approaches as well as using a dual formulation of the problem to prove the optimality of the solution. Similar to the machine learning mapping approach, knowledge of the task run-time is required.

2.4. Analysis

In conclusion, I consider my reading of the literature in terms of the key features that I argue a domain specific approach enables for heterogeneous computing: portability, prediction and partitioning

2.4.1. Portability

Abstraction and Modularity

The work on general purpose heterogeneous computing suggests that there has been significant progress in supporting execution upon heterogeneous computing systems. Such work takes the form of both proprietary tools, such as NVIDIA's CUDA, and open tools such as Harmonic, as well as tools that are platform specific, such as Vivado HLS, and those that support a wider range of systems such as IBM's Liquid Metal. Furthermore standards such OpenCL [14] and frameworks such as Liquid Metal [1] suggest that its possible to execute the same task description on multicore CPUs, GPUs and FPGAs.

Furthermore, there has been considerable research on the automatic mapping of tasks to heterogeneous platforms, including the work of Luk et al [17], Braun et al [40], Grewe and O'Boyle [44], Tarplee et al [53], Sajjapongse et al [55], Augonnet et al [57], Wang et al [58] and Beaumont and Marchal [56] to name but a few. These works describe many different strategies for mapping, including heuristic, machine learning and MILP-based approaches.

I build upon these tools, languages and standards to investigate how domain knowledge can be applied task to platform mapping so as to achieve the super-linear performance scaling described in Chapter 1. This means that any implementation effort upon my part is in interfacing with these tools, and this is a relevant design consideration. However, when building upon pre-existing tools and standards, similar levels of performance should be achieved as to those reported in the literature, comparable to those achieved by a programmer directly.

Hence, internal characteristics such as strong parallel scaling should exist as well as external comparisons to hand tuned implementations upon the same or similar platforms.

The Practical Value of Semantics

The end user programming work by Nardi [28] highlights that careful delimitation of the application domain can greatly reduce the functionality that has to be supported without overly inhibiting programmers.

Nardi's work also suggests that within a domain, the size of the subset of domain operations supported is prone to the law of diminishing returns - as a heuristic, there are 10-15 high level functions that are overwhelmingly used, and hence should be focused upon. The implications for my dissertation are clear, that with careful definition of the application domain being considered, I can potentially address a useful subset of a domain while meeting the needs of a number of domain programmers.

Another design consideration is the separation between the semantic model being considered and the implementation produced. As there are few formal and complete definitions of application domains, provided that I am clear on what is supported, I do not overly concern myself with capturing all of the semantics of the domain.

Only Portable Performance

The existing work in domain specific, heterogeneous computing [4, 34, 36] makes for encouraging reading, demonstrating good or better performance on a range of heterogeneous computing platforms compared to the platform code produced by programmers. Hence, these three independent bodies of work suggest that it is indeed possible to implement systems that translate domain specific inputs into executions for heterogeneous computing systems.

However, these bodies of work generally focus on supporting efficient execution, which suggests that there is space to consider how domain specificity can be applied to the other challenges that make heterogeneous computing inaccessible, such as workload partitioning. If domain programmers are able to execute domain specific tasks upon heterogeneous computing systems, they will require support in making efficient use of these resources. Beyond efficient implementations, this support requires a means to navigate the large design space that is enabled through multiple, diverse computing systems.

2.4.2. Prediction

Task-Platform Characterisation Process

It is clear that characterising a task upon a platform is a *complex* problem, in that it is defined by the interactions between the characteristics of both. As illustrated by the work on modelling approaches [49, 51], a further consideration is the fidelity requirements of the prediction. In many cases the prediction need not be perfectly accurate, but does need to be sufficiently close to reality to allow decisions to be made upon its basis, such as deciding between two platforms.

Hence, for my work, I need a means for characterising tasks with respect to platforms that is both repeatable and sufficiently accurate to achieve a useful end, such as super-linear performance scaling or helping programmers explore the heterogeneous computing design space.

Characterisation as a standalone activity

I found comprehensive literature describing the process of the characterisation of tasks upon heterogeneous platforms, with examples including the ASPEN [49], Sniper [48], Rodinia [47] and OP2 predictive modelling [60] work. This literature indicates that it is practically possible to characterise the performance of heterogeneous platforms.

The MODACloud [50] and the VEX/JINE [61] work goes further, explicitly performing the characterisation, but then using it in task mapping. Common is the specific characterisation of a task or group of tasks upon a platform for some particular purposes, such as supporting the claim that an implementation or platform is superior to others.

By keeping the characterisation process distinct from the mapping process, I can assess the characterisation process itself, and hence, qualify its potential impact upon the mapping process.

2.4.3. Partitioning

Preponderance of Heuristic Approaches

Generally heuristic approaches have been the most studied approach to partitioning work across platforms. Braun's comprehensive study [40] found that simpler heuristics achieve better results than more complex ones for the general case. This suggests that the truly optimal approach is case-specific, dependent upon the aforementioned complex dynamics between the task and platforms concerned, and so the more specialised a partitioning approach, the more likely it is to map better to certain configurations than others.

The well-founded heuristic approaches in the literature [40] suggest good starting points in partitioning work between heterogeneous computing resources. However, as these approaches are all founded upon human intuitions, they will invariably reflect human biases. An example is the *min-min* heuristic, the best found by Braun, where the quality of result depends on the ordering of tasks being considered. It is easy to *assume* that computational capabilities will be consistent across the platforms being available as is required by this heuristic, however the existence of super-linear performance scaling suggests otherwise.

Under-use of Integer Linear Programming

Integer Linear Programming appears to be an understudied approach although it is starting to see some attention [53], previously applied being applied in environments of pressing resource constraint [41, 59]. This lack of attention is likely due to the NP-hard complexity of integer linear programs in general, and NP-complete in the binary case. Bixby's retrospective work [62, 63] on the considerable progress made in linear and integer programming over the past two decades provides further insight as to why these approaches are underused. Until recently it wouldn't have been practical to use these approaches for partitioning problems of the scale of practical workloads. A insight from Bixby is that if an external measurement of solution quality exists, then a high quality solution that is not necessarily provably optimal can be derived in a tractable time.

MILP optimisation provides a rich toolbox with which to tackle the mapping problem, which, as the literature has shown, has proven itself resilient to resolution. If I seek to adopt such an approach, then the non-deterministic property of these methods does need to be addressed. This can be done by considering a suitably representative set of scenarios as well as its quality against other approaches.

3. Methodology

In this chapter I describe and motivate the methodology that fulfils my thesis that heterogeneous computing can be made more accessible through a domain specific approach. I also outline the criteria upon which I assess my methodology.

I explain this methodology in terms of the three features that I argue domain specificity provides:

1. **Portable Execution**: by focusing on a handful of influential functions within a domain, support for executing these functions across a wide range of platforms can be provided. Beyond providing an intuitive abstraction for using heterogeneous platforms, my approach can enable the optimal execution by drawing upon domain knowledge.

2. **Predictive Modelling:** automatic means for predicting quantitative characteristics or metrics. These metric models thus allow for the design space of a particular task upon a particular platform to be represented to the programmer in a domain form.

3. **Partitioning of Workloads**: functions for combining multiple task metrics upon one platform, and multiple platform metrics to a single value. This enables the expression of the partitioning of work as an optimisation problem with a single goal that is a unified representation of all tasks and all platforms.

This methodology is iterative in the sense that each feature is dependent upon the proceeding one. The first feature, portable execution, is based upon the concept of domain specific abstraction itself. As a result, the existence of each feature is itself evidence of the usefulness of the feature upon which it builds.

In this chapter I describe this methodology in general terms, as well as two small example domains, from the areas of image processing and linear algebra. In Part II, the Case Study, I use the example domain of derivative pricing to illustrate and evaluate this methodology more fully.

3.1. Portable Execution

In this section I describe how a domain specific approach can enable efficient execution on a range of heterogeneous architectures, without requiring additional effort on the part of programmer, beyond specifying the platform upon which the task should be executed. The key enabler of this feature is provided by the domain specific abstraction: by limiting the operations allowed to a small set, it is easier to implement uniform support upon a range of heterogeneous platforms.

I start by assuming that the domain semantic model exists, such as the derivative pricing one that I detail in Section 4.2. I show how from the semantic model the *domain data types*

Domains	Image Filtering	Linear Algebra Arithmetic
Data types	*Images, Kernels*	*Matrices, Vectors, Scalar*
Functions	*Apply*	*Addition, Subtraction, Multiplication, Division*

Table 3.1.: Domain data types and functions examples

and *domain functions* can be extracted. These types and functions are the concepts and corresponding syntactical constructs that the programmer can use to express their computation as *domain programs*. I then describe the necessary supporting infrastructure for executing domain programs on heterogeneous platforms. Finally, I expand upon how domain programs can be formulated such that efficient execution is ensured on heterogeneous platforms. These benefits are derived from the dependencies between the domain data types that can be made explicit in the data type definition, as a result of the parallelism that can be extracted from common domain functions *a priori*.

3.1.1. Domain Data types and Functions

Domain data types and functions are the categories that I use to describe the implementations of the abstractions defined in the domain's semantic model. In object orientated terms, the semantic model provides the prototypes for the data type and function classes. The distinction between the two is a matter of semantic preference - I prefer to make the distinction between data types, the attributes of the classes in the semantic model, and functions, the methods of the classes, when describing the classes that comprise the domain. I find that this distinction mirrors the *noun-verb* formulations of problems found in application domain terminology. However, equally, an object-orientated class could be used to capture both data types along with the functions (or methods, strictly speaking) associated with that data type.

For example, in the domain of filtering within the larger area of image processing, as given in Table 3.1, two domain data types might be the *image*, the data structure to which filters may be applied, as well as *kernel*s, the data structure representing the weighting of the filter convolution operation. A domain function would be *apply*, that takes both a *kernel* and an *image* as an argument and returns a modified *image*, having performed the convolution of *kernel* and *image*. Alternatively, a *filter* class might be defined that encapsulates both the *kernel* as attributes, and *apply* as a method, The *apply* method takes an *image* class instance as an argument, and returns a modified *image* instance.

A mathematical formulation of domain data types and functions is provided in (3.1). Where a domain function (F) defines a mapping from a set of domain data type inputs (P) to a domain data type result (R) . Both P and R belong to a larger set of domain data types ($\mathcal{D}$).

$$F : P \mapsto R \quad P, R \in \mathcal{D}. \tag{3.1}$$

Domain Data types

Domain data types are the fundamental concepts within the application domain's semantic model, i.e. the conceptual information objects that are *atomic* in the sense that considering a finer level of granularity in isolation would be nonsensical in the domain context[1]. Compared to general purpose primitive types, domain types are usually at a much higher level of abstraction. For example, in the linear algebra arithmetic domain, matrices, vectors and scalar values would all be examples of domain data types, as given in Table 3.1.

A useful feature of domain data types is that the relationships between different instances can be made explicit in the type definition, captured in common patterns such as collection, containment or producer-consumer. As I show later, such dependencies can then be used to balance communication and computation statically. Continuing the linear algebra arithmetic example, a *system* data type could be introduced that contains all the *matrix, vector or scalar* instances that are going to be used in a given set of arithmetic of operations.

Domain Functions

Domain functions are transformations or operations that may be applied to domain data types, that return a result that is meaningful in the context of the domain, i.e. another domain data type instance[2]. In the arithmetic linear algebra, examples of domain functions would be arithmetic operations such as addition, subtraction, multiplication and division as defined for matrices, vectors and scalar types.

As the structure of the domain functions have to be predefined, multiple implementations of the same function can be analysed in advance, and optimal configurations for different architectures may be found. This flexibility in the function definition also has benefits at run-time, providing scope for the system to automatically modify a particular implementation safely, without affecting the correctness of the result. To borrow terminology from operations research, the domain function definition clearly delineates the *parameters* of the function, the domain data type input instances, which cannot be altered, and the *variables*, implementation variables that can.

3.1.2. Supporting Infrastructure

While the previous section has described the application domain as it would be viewed by the domain programmer, this subsection discusses the bridge between those domain abstractions and actual implementation upon heterogeneous computing platforms. In doing so, I first consider the means of expression, compilation framework, and the run-time flow required for a domain specific approach. In this subsection, I am describing the second and third stages of the process depicted in Figure 2.2, as adapted for heterogeneous computing.

The domain specific approach I propose is a black box application framework [30]. Beyond using the data types and functions made available to them, the domain programmer has no control over the implementation of the program that they have specified. In many contexts such a limitation would either not noticed, or would be acceptable. By abstracting away much of the control from the domain programmer, the system programmer, as described in Section 2.2.1, is given greater freedom to imbue the system with the ability to implement efficient execution

[1]For example, an integer value, representing the colour intensity of a pixel, would make sense in the image processing domain, but arguably a standalone scalar value does not

[2]This does not exclude the existence of domain functions that take no inputs, for example operations that generate data

Listing 3.1: Example of image filtering external DSL, similar to Halide [64], applying a blur filter without locality of execution defined

```
load source_image_inst from 'fabio.jpg'
kernel_inst is [[1/9,1/9,1/9],[1/9,1/9,1/9],[1/9,1/9,1/9]]
apply kernel_inst to source_image_inst as dest_image_inst
save dest_image_inst as 'fabio_blurred.jpg'
```

automatically. As I show in the next subsection, 3.1.3, this efficiency comes from making use of prior knowledge of both data type dependencies and the structure of the domain function.

As described in Section 2.2.2, the system programmer needs to make three design decisions when implementing a domain specific, heterogeneous system: the syntactical form of the domain specific program, the depth of the compilation process from program to executable, and finally the nature of the executable.

An approach I don't consider is the interpretation of domain programs' source code as opposed to compilation. Beyond multicore CPUs, any approach that would interpret domain programs for execution upon heterogeneous platforms would require a precompiled library of significant sophistication, beyond that which can be provided practically in most domains. A case in point of this analysis is graphics domain programming for GPUs, which requires not only vendor-provided operating systems drivers, but also widely adopted standards such as OpenGL.

Means of Expression

The first design choice that the system programmer must make is between using an external or internal DSL, or an application framework as the means by which the domain programmer implements their computation. In the context of heterogeneous computing, the system programmer must carefully delineate between what functionally will be supported upon heterogeneous platforms versus what can be trivially implemented using a conventional programming environment. A further consideration is the degree to which this the locality of execution is transparent to the domain programmer.

To use the example of image filtering, as given in Listing 3.1, the system programmer might define an external DSL, similar to Halide [64], that in addition to the image and kernel data types and the apply function, provides various support functions for loading, converting and saving images from and to popular image file formats. It would be within reason (and prudent) for the system developer to note in the documentation that the loading and conversion functions are only supported on conventional CPUs[3], but not define where the apply function will be implemented, leaving it up to the domain specific system.

As a contrasting example, given in Listing 3.2, a linear algebra arithmetic application framework, similar to BLAS [32], has been defined that provides platform specific domain data types and functions.

[3]Not in the least because to do otherwise would enatil support for file systems on heterogeneous platform such as GPUs and FPGAs, considerable research topics in their own right!

Listing 3.2: Example of linear algebra arithmetic application framework in C, similar to BLAS [32], performing vector-scalar multiplication upon a GPU.

```
vector_t vector_inst = {5,1,2,3,4,5};
scalar_t scalar_inst = 2.0;
vector_t result = laa_api_gpu_vsm(&vector_inst,&scalar_inst);
```

Compilation (and Code generation):

The next choice for the system programmer is the depth and complexity of the compilation process, as outlined in Section 2.2.2. The compilation must take the domain program and create an executable that can run on the available computing resources. Considering the arithmetic linear algebra code in Listing 3.2, even straight-forward vector-scalar multiplication could be compiled into multiple, functionally equivalent codes based upon analysis of the input data. For example, the domain specific system could detect whether the vector specified was of a sufficient degree of sparsity to change from a dense algorithm to one which uses a compressed data representation.

In the heterogeneous computing context, code generation is an attractive approach, as it allows for a clear separation between the domain programming environment and the heterogeneous implementation, allowing for arbitrary complexity in the former and the simplicity often required by exotic platforms in the latter. In the code generation approach, code can generated from the domain program in a form, such as OpenCL, that can then be used an input in a heterogeneous platform compilation flow.

In Listing 3.3, generated OpenCL code for the blur filter function as per the DSL in Listing 3.1 is given. The code, that could be run upon multicore CPUs, GPUs or FPGAs, is simple. In addition to using declared constant, all of the image loading, converting and saving is performed in the host code, all image data has been reordered into contiguous elements in the input and destination arrays.

Execution

Finally the compiled domain executable must be made available to the domain programmer. This can either be as a standalone executable program binary that can be executed within a conventional operating system environment, or from within the domain specific environment, where the execution of the executable is managed by the domain specific heterogeneous system. The latter is useful in obscuring low-level communication or configuration of heterogeneous computing resources that must occur in advance or prior to execution.

For example, the linear algebra arithmetic framework in Listing 3.2 might require that an appropriate header file is included in the required C code, and that an initialisation function is called prior to use, and shutdown function after use, as given in Listing 3.4:

3.1.3. Using domain knowledge

This section describes how knowledge from the application domain can be used to make execution upon heterogeneous platforms more efficient without programmer intervention. First, I show

Listing 3.3: Generated OpenCL code for image filtering blur example

```
kernel void image_kernel_apply(
  const global float *source,
  const global int source_width;
  const global float *kernel,
  global float *dest
  ){
  //Assuming kernel dimension N has been defined at compilation
  int i;

  //finding the upper corner of the filter window
  int x = get_global_id(0) + N/2;
  int y = get_global_id(1) + N/2;

  int start_x = x - N/2;
  int start_y = y - N/2;

  //computing the result
  float result = 0;
  KERNEL:
    for(i=0; i < N; ++i)
      for(j=0; j < N; ++j)
        result +=
          kernel[i * N + j] *
          source[(start_x+i) * N + start_y + j];

  //outputting the result
  dest[x * source_width + y] = result;
}
```

Listing 3.4: Example of full linear algebra arithmetic application framework program in C

```
#include "laa_api_gpu.h" //Linear Algerbra Arithmetic GPU Library

int main(void){
  laa_api_gpu_init(); //device initialisation

  vector_t vector_inst = {5,1,2,3,4,5};
  scalar_t scalar_inst = 2.0;

  //GPU vector-scalar multiply
  vector_t result = laa_api_gpu_vsm(&vector_inst,&scalar_inst);

  free(vector_inst);
  laa_api_gpu_shutdown(); //device shutdown
  return 0;
}
```

how the domain data types can be defined to make dependencies explicit, and hence make it easier to balance computation and communication. I then show how task, data and pipeline parallelism, can be extracted from function structure ahead of execution.

Relationships between data types

One of the challenges in large scale, parallel heterogeneous computing is balancing the time spent performing computational operations versus time spent communicating data to and from the computational platforms within the system [65]. Ideally, communication must be localised, so as little time as possible is spent performing the communication, and regularised, so computations can be scheduled to make the most efficient use of the computational platforms.

As mentioned in Section 3.1.1, the definition of domain data types can aid in this balancing by making the relationships between data type instances a defined property. This means that in the worst case, the memory bandwidth of the computation will be the fastest memory resource that can accommodate the related data type instances. At a finer granularity, such information could be used to improve the efficiency of the memory hierarchies of the heterogeneous platforms.

For example, in the image filtering case, an image instance might have a filter instance associated with it, which the domain specific system could use to infer that the two should be collocated in the memory of a platform. At the finer level, as can be seen in Listing 3.3, the system can preload the kernel instance many times into local efficient, constant memories on massively parallel platforms, whilst "chunking" the image instance according to the size of the kernel, and storing it in a more convenient form in global memory.

Function Structure

As the structure of domain functions is known in advance, the parallelism in these functions can be identified and used in the heterogeneous implementations. As many heterogeneous platforms, such as GPUs and FPGAs, have considerable parallel compute capability, functions in a form suitable for parallel execution can take advantage of this.

The domain functions can be in a parallel form because if the result returned matches that which is specified in the semantic model, the system is free to implement these operations in whichever form is most efficient. The three forms of parallelism I consider are task and data parallelism, and pipeline parallelism.

Task Parallelism or Multiple Instruction, Multiple Data (MIMD) [66] is achieved by performing multiple tasks in parallel, i.e. at the same absolute time. As the structure of domain functions are known in advance, opportunities for task parallelism can be identified, and the compilation and run-time environments can be configured to exploit this. Exploiting task parallelism is widespread, thanks to the popularity of software libraries such as OpenMP [67], Pthreads [68] and Threaded Building Blocks [69].

For example, the arithmetic linear algebra domain is rich with opportunities for parallel execution. Due to the linear nature of arithmetic operations, all independent arithmetic terms could be computed in parallel. Listing 3.5 gives an OpenCL kernel for the GPU vector scalar multiply API function used in Listing 3.4. This kernel could be executed in a task parallel fashion

Listing 3.5: OpenCL kernel code for vector scalar multiply example

```
kernel void laa_api_gpu_vsm_kernel(
  const float *vector,
  const float scalar,
  global float *result
  ){
  int i = get_global_id(0);

  //Scaling this particular vector value
  result[i] = vector[i] * scalar;
}
```

by specifying multiple work-groups within the global work-set. The result of specifying multiple work-groups on a multicore CPU system would result in each core of the CPU performing batches of scalar-vector element multiplies. Within each core, the multiplies allocated to it would be performed sequentially.

Data Parallelism or Single Instruction, Multiple Data (SIMD) [66] is achieved by performing the same operation upon multiple instances of data. Similar to task parallelism, as the structure of domain functions is known *a priori*, opportunities for data parallelism can be made explicit to the compilation and run-time systems.

For example, keeping with the arithmetic linear algebra domain, each vector arithmetic operation is itself composed of many, identical scalar arithmetic operations. In OpenCL, the example in Listing 3.5 could be executed in a data parallel fashion by specifying the work-groups, such that there are multiple work-items within each work-group. In a GPU, unlike the multicore CPU in the previous example, multiple work-items in a work-group could be evaluated at the same time, by applying the same operation to multiple elements in a lock-step fashion.

Pipeline Parallelism is achieved by multiple elements in a dataflow working in parallel, upon different stages of the computation. Domain specific approaches are well suited to exposing pipeline parallelism, as by knowing the structure of functions in advance, as well as knowing about the dependency relationships between data instances, the schedule of pipeline elements can often be predicted in advance[4]. Execution environments can take advantage of this by maximising the use of the available resources. A key consideration in exploiting pipeline parallelism is ensuring that there is adequate memory resources buffering between pipeline stages so that pipeline stages performed at different rates do not become stalled by a bottleneck in the pipeline.

For example, in the image filtering case, a computational pipeline could be built by unrolling the KERNEL loop defined in Listing 3.3, with a stage for each weight in the kernel. The data in the image could then be "streamed" through the computational pipeline, with each cycle resulting in a new result for the filter operation.

[4]The use of the dataflow paradigm in OpenSPL [15] is for the same reason.

3.1.4. Portability Criteria

To prove the property of portable, efficient execution, I first need to provide implementations across a wide range of diverse architectures, for example multicore CPUs, GPUs and FPGAs, to suggest that the domain abstraction can easily enable heterogeneous execution. Secondly, I need to demonstrate that these implementations are efficient, using evidence from the platform run-time, as well as comparisons with external, expert programmer implementations.

A consideration in evaluating this feature is that its contribution is not novel, as described in Section 2.4.1. In addition to the work described in this dissertation, this portable execution feature has been demonstrated in practice by the work described in Section 2.2.3.

3.2. Prediction of Run-time Characteristics

While the previous section addressed how a domain specific abstraction can enable portable, efficient execution upon heterogeneous platforms, in this section I consider the nature of that execution. In the same way that domain specific abstractions exist for computations, I assert that application domains also provide abstractions for measures of quality or *domain metrics*. These metrics qualify the domain data type result from the computation, in measures of performance and quality using domain terminology.

While (3.1) presented the domain function completely abstracted from execution context, in (3.2) I introduce the context: a domain function (F) defines a mapping from domain data type parameters (P) and implementation variables (V) inputs to domain data type results (R) and domain metric outputs (M).

V is in the implementation variable set ($\mathcal{V}$). V includes all non-domain values which determine how F is implemented on a particular platform, for example algorithmic parameters which have no meaning in the domain context.

M is in the domain metric set ($\mathcal{M}$). M is the values which qualify R in quantified measurements of performance and quality that have meaning in the domain.

$$F : (P, V) \mapsto (R, M) \quad V \in \mathcal{V},\ M \in \mathcal{M},\ P, R \in \mathcal{D}. \tag{3.2}$$

While P is the input as specified by the domain programmer, and R the output that they seek, M is of interest to the domain programmer, as it relates R with respect to F. The values of M are the result of the interaction of the parameters specified by the domain programmer, P, and V which can be specified by domain programmer, or as I propose, by the domain specific system.

The capability to predict domain metrics is the second feature of the domain specific approach, as it allows for the useful characterisation of heterogeneous platforms. By useful characterisation, I that the domain specific approach enables predictive modelling of the metrics of domain functions. Such a characterisation is useful as it allows for the comparison of different platform implementations before execution. This characterisation is a natural extension of the previous, portability feature, which provided domain specific abstractions for task execution, this prediction feature provides a domain specific abstraction for task execution on platforms.

The domain specific characterisation relates tasks, as represented by the domain data types and functions, and platforms in terms of the domain abstraction. Furthermore, by modelling the task-platform relationship that results from varying implementation-specific variables in domain metrics, a computational design space can be made accessible to the domain programmer. I argue that the relationship between implementation variables, as represented by domain metrics, is best represented as a Pareto surface. This Pareto surface captures the design space trade-offs that exist for a particular task upon a platform. Providing such a representation allows domain programmers to balance their objectives for themselves, instead of the status quo, where the balance is determined on the whim of the system programmer.

I outline this prediction feature by first describing the nature of the domain specific task-platform design space, as defined by the implementation variables. I then show how general models that relate the inputs to domain functions to domain metric values can be found. Finally I describe how the general models for domain functions can be specialised to specific domain program-platform instances.

3.2.1. Domain Parameters, Implementation Variables and the Design Space

As given in (3.2), ($P \in \mathcal{D}$) and ($V \in \mathcal{V}$) contain all possible inputs to the domain function. As described in the previous subsection, from the perspective of the domain specific system, the values of P are immutable parameters.

For example, in Listing 3.3, if the system was to change the values of the kernel weights in any way, the result would be invalidated. However, in the OpenCL kernel code in Listing 3.6, a compiler definition option, PRIVATE_MEMORY, has been added that instructs the OpenCL compiler to use private memory resources to be used in a pixel calculation. From the perspective of the domain programmer, the result, R, will be the same if this option is used or not. Hence, the private memory option is an *implementation variable*, as it could be modified without affecting the domain result. This illustrates the key property of implementation variables - these variables do not have to be visible or mutable by the domain programmer.

This definition implies that the domain parameters, P, and by extension, the domain abstraction, provide a definition of correctness for the domain function. Hence, P provides a means to partition all possible inputs into the domain functions, F, into valid and invalid inputs. All valid inputs to F must contain P, i.e. P with any V defines the domain design space. This definition is immediately useful, as it opens the door to automated exploration of the domain design space.

At its simplest, a system could explore all of the members of $\mathcal{V}$, and then select the implementation that achieves the goals of the system programmer, or the presumed goals of the domain programmer. However, even with a modest number of implementation variables with reasonable bounds, such a 'brute force' enumeration of the design space quickly falls prey to the curse of dimensionality.

Listing 3.6: Generated OpenCL code for image filtering blur example

```
kernel void image_kernel_apply(
  const global float *source,
  const global int source_width;
  const global float *kernel,
  global float *dest
  ){
  //Assuming kernel dimension N has been defined at compilation
  int i;
  int x = get_global_id(0) + N/2;
  int y = get_global_id(1) + N/2;

  //Finding the upper corner of the filter window
  int start_x = x - N/2;
  int start_y = y - N/2;

  float result = 0;

  #ifdef PRIVATE_MEMORY
  //Copying source to private memory
  float p_source[N*N];
  KERNEL:
    for(i=0; i < N/2; ++i)
      for(j=0; j < N/2; ++j)
        p_source[i * N + j] = source[(start_x+i) * N + start_y + j]

  //computing the result using private memory
  for(i=0; i < N*N; ++i)
    result += kernel[i] * p_source[i]

  #else
  //computing the result
  KERNEL:
    for(i=0; i < N; ++i)
      for(j=0; j < N; ++j)
        result +=
          kernel[i * N + j] * source[(start_x+i) * N + start_y + j]
  #endif

  //outputting the result
  dest[x * source_width + y] = result;
}
```

Domains	*Image Filtering*	*Linear Algebra Arithmetic*
Latency	Seconds	Seconds
Throughput	$\frac{\text{Images}}{\text{Second}}$	$\frac{\text{Matrices}}{\text{Second}}$
Quality	Decibels	Unit of Least Precision
Resource Use	$\frac{\text{Images}}{\$}$	$\frac{\text{Vectors}}{\$}$

Table 3.2.: Domain metric unit examples for image filtering and linear algebra arithmetic domains, using the data types given in Table 3.1.

3.2.2. Domain Metrics and Pareto Optimality

Domain Metrics

The previous subsection defined the design space of the domain function, but for this space to have meaning to the domain programmer, a means to describe the space in terms of the domain is required. Domain metrics provide such a means, using a measure of the achievement of a goal commonly held within the domain. These domain metrics fall into one of four categories:

- Latency - the absolute time between task initiation and completion.
- Throughput - the average rate at which tasks are completed.
- Quality - the measurable degree to which a task achieves the goal of the domain program.
- Resource Use - the degree to which the task is using the available resources.

In Table 3.2, an example of a unit for each category of metric is given for the example domains of image filtering and linear algebra arithmetic. To find the computational design space for a task within an application domain, without using a brute force exploration, a model is required for the mapping of the task implementation variables to domain metrics on the target platform.

To show how models for domain metrics can be found, in (3.4), I have repeated (3.2)'s formulation, but with domain metric outputs (M) in addition to the domain data type output (R). Furthermore, the input and outputs sets have been defined in (3.3) as real-valued vectors, as would mostly likely be the case in practice.

$$\mathcal{P} = \mathbb{R}^p,\ \mathcal{V} = \mathbb{R}^v,\ \mathcal{M} = \mathbb{R}^m,\ \mathcal{R} = \mathbb{R}^r, \tag{3.3}$$

In (3.4), I show the mapping of p domain parameter and v variable inputs to r domain result and m domain metric outputs, where $\vec{F}$ is the vector form of the domain function, $(\vec{P}, \vec{V})$ the inputs and $(\vec{R}, \vec{M})$ the output value vectors. $\vec{F}$ is composed of at least m projection functions, f_k, each of which map $(\vec{P}, \vec{V})$ inputs to a single metric value. To model the metric outputs of $\vec{F}$, models for f_k need to be found.

$$\begin{aligned} \vec{F} = (f_1, f_2, \cdots, f_m) : (\vec{P}, \vec{V}) \mapsto (\vec{R}, \vec{M}) \quad & \vec{P} \in \mathcal{P},\ \vec{V} \in \mathcal{V},\ \vec{M} \in \mathcal{M},\ \vec{R} \in \mathcal{R}, \\ f_k(\vec{P}, \vec{V}) = M_k \quad & k = 1, 2, \ldots, m. \end{aligned} \tag{3.4}$$

(3.2) captures the functional description of the domain task, i.e. how it maps the domain data types inputs in outputs. (3.4) goes beyond this, providing a contextualised description of

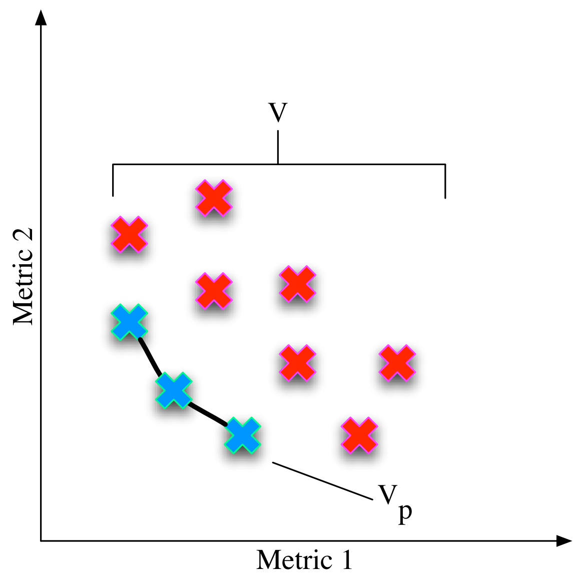


Figure 3.1.: Diagrammatic representation of Pareto Optimality in the context of the domain specific methodology. This case assumes it is desirable to minimise both metrics.

the domain task. By contextualised, I mean that the additional domain metrics outputs qualify the results of the computation, providing the domain programmer with additional information. This provides the relationship between the domain task and the computing platform being used to perform it.

As the application domain identifies in advance those domain functions which are disproportionately used, as described in Section 2.4.1, hence a model function for mapping $\vec{P}, \vec{V}$ to $\vec{M}$ of those key domain functions can be found prior to execution. Thus the model ($\vec{F}$) for the most important functions in a domain can be found in advance.

Using Metrics to define Pareto Optimality

As domain metrics provide the means to characterise implementation in domain terms, in (3.5) I define that the set of implementation variables can be partitioned into two disjoint subsets, Pareto optimal ($\mathcal{V}_p$) and non-Pareto optimal ($\mathcal{V}_n$) input values.

$$\mathcal{V} = \mathcal{V}_n \cup \mathcal{V}_p \quad \mathcal{V}_n \cap \mathcal{V}_p = \emptyset. \tag{3.5}$$

I have defined $\mathcal{V}_p$ in (3.6) and illustrated it in Figure 3.1. I define Pareto optimal implementation variables as those that optimise at least a single value of M, as defined in the domain. $\mathcal{V}_{np}$ defines all of those inputs which do not. While $\mathcal{V}$ constitutes the domain design space, $\mathcal{V}_p$ is the Pareto optimal design surface.

$$\forall \vec{x} \in \mathcal{V}_p \quad \forall \vec{y} \in \mathcal{V} - \vec{x} \quad \exists k \in 1, 2, \ldots, m \qquad f_k(\vec{x}) \prec f_k(\vec{y}) \tag{3.6}$$

To illustrate how domain metrics allow for the Pareto optimal implementation variables to be found, I return to the vector scalar multiply example. In Listing 3.7 a functionally equivalent kernel to Listing 3.5 is given. However an additional implementation variable, CHUNK, has been specified, defining the number of elements that should be evaluated within each OpenCL work-

Listing 3.7: Generated OpenCL code for vector scalar multiply example

```
kernel void laa_api_gpu_vsm_kernel(
  const float *vector,
  const float scalar,
  global float *result
  ){
  int i = get_global_id(0)*CHUNK, j;
  float p_vector[CHUNK], p_result[CHUNK];

  //Copying data to the device's private memory
  float p_scalar = scalar;
  for(j=0;j<CHUNK;++j)
    p_vector[j] = vector[i+j];

  //Performing the computation
  for(j=0;j<CHUNK;++j)
    p_result[j] = p_vector[j]*p_scalar;

  //Writing the result
  for(j=0;j<CHUNK;++j)
    result[i+j] = p_result[j];
}
```

item. In addition to allowing the overhead per work-item being amortised over multiple vector elements, there might be a performance benefit in grouping global memory accesses together[5]. However, if a large enough chunk size is specified, lower cache hit rates might occur, and hence increase the latency of memory accesses.

Hence, a potential latency model as a function of CHUNK size is given in (3.7). The model function ($\tilde{f}_l$) is parabolic in nature, reflecting the second order effect of caching on latency, with the value of CHUNK (N_C) that gives the minimum a member of the Pareto optimal domain variable set ($\mathcal{V}_p$). α, β and δ are constants that reflect the vector size and various platform characteristics. The model also assumes the size of the vector in question will be much greater than N_C.

$$\tilde{f_L}(N_C) = \alpha(\beta - N_C)^2 + \delta \tag{3.7}$$

The domain knowledge of what metrics matter, and hence should be modelled, is of crucial importance in exploiting heterogeneous computing resources. The effort of doing so is returned with interest, as domain metrics provide an intuitive way of understanding heterogeneous platforms for the domain user. By creating models of the relationships between domain metrics as achieved through different configurations, the achievable design space is provided in a form that the domain user both understands and can reason about in light of their own objectives.

[5]Many compilers will transform the first and third loops into a single memory access

3.2.3. Implementing Domain Metric Models

Identifying

The formalism described in Section 3.2.2 helps identify the criteria for potential model functions, however for each domain function there are an infinite number of possible metric model functions. Similar to providing platform-specific implementations of disproportionately used functions, in this methodology I require the system programmer to find appropriate domain metric models.

When identifying metric models, I found Occam's Razor to be a useful heuristic, and hence I always opted for the simplest possible polynomial model that was able to predict the metric value for the domain function as a function of the implementation variables.

Populating

As the structure of $\vec{F}$ is deterministic, an online benchmarking approach can be used to gather data for input into a weighted least squares regression. Weighted least squares can then be used to solve for the task and platform-specific metric model coefficients. I propose a three step process for populating domain metric models that will represent the design space:

1. *Benchmarking* - a subset, $\mathcal{B} \in \mathcal{V}$, of the implementation variables are executed and the domain metrics that are to be modelled, are measured. The result of this benchmarking for the vector case is a tuple of implementation variables and metric values, i.e. $(\mathbb{R}^{b\times v}, \mathbb{R}^{b\times m})$, where b is the number of values in $\mathcal{B}$. The length of benchmarking activity would be determined by a heuristic value of what is "reasonable" to the domain programmer, such as a few minutes.

2. *Solving* - once the initial subset of the task has been completed, the results can be used to solve for domain metric model coefficients using weighted least squares regression. A regression technique is useful in this context, as it is able to not only accommodates natural degrees of uncertainty that arises in any complex system, but also can compensate to some degree for the error within the structure of the model itself.

3. *Prediction* - after the model coefficients have been found, the algebraic models can predict the domain metrics as determined by the implementation variables. These algebraic models could be used to create representations of the metrics that the domain programmer may interact with. I believe that a Pareto curve or surface is a natural representation of this information, as it would allow the domain programmer to trade between various metrics using graphical interfaces.

It should be remembered that this process is in aid of guiding the domain programmer - if a prediction turns out to diverge strongly from reality, the system could always halt execution and inform the programmer of this. Furthermore, as the problem execution is underway, the metric model could be updated, providing more accurate feedback to the domain programmer.

3.2.4. Predictability Criteria

To provide useful predictions as described in Section 2.4.2, I suggest that a domain metric model needs to have two properties:

Incorporation: when provided with additional data points, i.e. a larger b, the predictions made by the models of domain metrics should converge on the true value of the domain metric that the model is predicting.

I have defined the relative error of a prediction in (3.8), where r_k is the relative error for domain metric k, f_k is the value of the metric as measured by the domain specific implementation and $\tilde{f_k}$ is the metric model.

$$r_k = \frac{|f_k(\vec{P}, \vec{V}) - \tilde{f_{k,b}}(\vec{V})|}{f_k(\vec{P}, \vec{V})} \tag{3.8}$$

I have given the convergence criteria in (3.9), where the benchmarking set converges on the set of all possible implementation variables, the relative error converges on a small, constant relative prediction error (ϵ_k), which reflects the small difference between the model and the actual implementation.

$$\lim_{\mathcal{B} \to \mathcal{V}} r_k \to \epsilon_k \tag{3.9}$$

Extrapolation: for a finite amount of benchmarking, the models should be able to make predictions close to ϵ_k for implementation variable values a considerable distance from those used in the benchmarking set. Heuristically, I have found prediction models need to be able to extrapolate for order of magnitude or greater difference, with an increase of error less than an order of magnitude in scale, given a starting $r_k < 0.1$.

3.3. Partitioning of workloads

While the characterisation described in the previous subsection is useful when considering how to use particular heterogeneous platforms in isolation or when selecting a platform exclusively from an array of platforms, it is less helpful when faced with a cluster of heterogeneous computing resources that can be used cooperatively.

In this subsection, I address the third feature of the domain specific approach - efficient workload partitioning. I show how multiple domain metric model functions can be combined so as to create a unified design space. I then introduce the key conceptual tool: the expression of the distribution of work as an optimisation problem using the metric models.

I first generalise the makespan minimisation problem from Section 2.3.2, integrating it with formalism developed earlier in this chapter. I then show how information from the domain can be used to increase the degree of distributed, parallel execution of domain tasks. Finally I describe how multiple metrics can be optimised, so that the domain specific Pareto surface may be found.

3.3.1. Generalising the Allocation Problem

I begin by deriving the general allocation problem from the makespan minimisation problem as described in Section 2.3.2. I can generalise this problem, making use of the notion of domain metric models given in (3.4) and the domain Pareto optimal implementation variables, as given in (3.5). I assume that the Pareto optimal variables $\vec{V_p}$ for each of the μ platforms are already known or can be easily approximated for each of the τ tasks[6].

In (3.10) I seek an allocation ($\boldsymbol{A}$) so that I optimise the metric k for all tasks, as projected by the task and platform reduction functions ($\vec{F_k}$) and $G_k(\boldsymbol{A}, \boldsymbol{P_p})$) into a scalar value. The binary elements of $\boldsymbol{A}$ indicate whether a task has been assigned to a particular platform, i.e. if $A_{i,j} = 1$, then task j has been assigned to platform i. Domain metric models can then be used to find the metric values for the assigned task-platform pairs.

However, to optimise a single metric using the allocation, a means for projecting the metric values to a single scalar value is required. Hence, I have introduced two *reduction functions*, G_k and $\vec{F_k}$ that play a vital role in enabling this formulation. Firstly, the task reduction ($\vec{F_k}$) reduces the metric values for multiple tasks upon multiple platforms to a single metric value per platform. Then, the platform reduction function (G_k) reduces the metric values for multiple platforms to a single scalar value. The nature of the projection being performed by these functions would be defined within the domain, for each metric. For example, in the case of the makespan, the task reduction function would sum the platform's tasks' latencies together, while the platform reduction function would find the platform with the greatest latency.

$$\begin{aligned} \underset{\boldsymbol{A}\in\{0,1\}^{\mu\times\tau}}{\text{optimise}} \quad & G_k(\vec{F_k}(\boldsymbol{A}, \boldsymbol{V_p})) \quad \boldsymbol{V_p} \in \mathbb{R}^{\mu\times\tau\times v}, \\ \text{subject to} \quad & \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1, 2, \ldots, \tau. \end{aligned} \tag{3.10}$$

where:

$$\begin{aligned} G_k &: \vec{M_k} \mapsto M_k, \\ \vec{F_k} &: (\boldsymbol{A}, \boldsymbol{V_p}) \mapsto \vec{M_k}. \end{aligned}$$

For example for a workload of vector scalar multiplication tasks with the chunking implementation variable(N_c), as described in the Section 3.2.2, the platform and task reduction functions for the latency function are given in (3.11), using the hypothetical metric model proposed in (3.7).

$$\begin{aligned} G_L(\vec{F}_L(\boldsymbol{A}, \boldsymbol{N}_C)) &= \max(\vec{F}_L(\boldsymbol{A}, \boldsymbol{N_C})), \quad \boldsymbol{N}_C \in \mathbb{Z}^{\mu\times\tau} \\ \vec{F}_L(\boldsymbol{A}, \boldsymbol{N}_C) &= (\boldsymbol{A} \circ (\alpha(\beta - \boldsymbol{N}_C)^2 + \gamma)) \cdot \mathbf{1}. \end{aligned} \tag{3.11}$$

3.3.2. Splitting the Atomicity of Tasks

Similar to heterogeneous execution and characterisation features, knowledge from the application domain can help find an efficient solution to the allocation problem. As the structure of domain functions is known *a priori*, the degree of parallelism within a task is known. As a result, partitioning approaches can incorporate this information so that a task can be divided into subtasks while still providing a correct result.

[6]I recognise that this is a rather dramatic assumption, however the area of autotuning, of finding optimal implementation variables automatically, is showing much promise. The work of Wang [70], Tournavatis [71] and others suggests automated means for finding implementation variables.

Making parallelism explicit enables a greater degree of work sharing between distributed computing resources, as discussed in Section 2.3. If the degree of parallelism is sufficiently large, i.e. the tasks are "embarrassingly parallel" in nature [65], the elements of the allocation matrix, $\boldsymbol{A}$, can be "relaxed" to be real-valued and the problem becomes linear, and hence more tractable, as expressed in (3.12).

$$\begin{aligned} &\underset{\boldsymbol{A}\in\mathbb{R}_{+}^{\mu\times\tau}}{\text{optimise}} && G_k(\boldsymbol{A},\boldsymbol{V_p}) \quad \boldsymbol{V_p}\in\mathbb{R}^{\mu\times\tau\times\upsilon}, \\ &\text{subject to} && \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1,2,\ldots,\tau. \end{aligned} \tag{3.12}$$

Such a relaxation would be appropriate in vector scalar multiply example - the degree of allocation would translate to the number of elements that were being processed on a particular platform. For example, if $A_{0,1} = 0.5$, the half of the elements in task 1 would be processed on platform 0.

In (3.13) I have given a vector of values for the latency model in the task reduction (3.11). In this example, a single task is being partitioned between two platforms.

$$\vec{F}_L(\boldsymbol{A},\boldsymbol{N}_C) = (\boldsymbol{A}\circ[\ 1 \quad 3\ ])\cdot\mathbf{1} \tag{3.13}$$

In (3.14), the optimal allocation is given if $\boldsymbol{A}$ is only allowed to be binary valued.

$$\begin{aligned} \boldsymbol{A} &= [\,1 \quad 0\,] \\ \therefore\quad \vec{F}_L(\boldsymbol{A},\boldsymbol{N}_C) &= ([\,1 \quad 0\,]\circ[\,1 \quad 3\,])\cdot\mathbf{1} \\ \vec{F}_L(\boldsymbol{A},\boldsymbol{N}_C) &= [\,1 \quad 0\,]\cdot\mathbf{1} \\ \therefore\quad G_L(\vec{F}_L(\boldsymbol{A},\boldsymbol{N}_C)) &= 1 \end{aligned} \tag{3.14}$$

In (3.15), the optimal allocation is given if $\boldsymbol{A}$ is allowed to be relaxed to be real valued.

$$\begin{aligned} \boldsymbol{A} &= [\,0.75 \quad 0.25\,] \\ \therefore\quad \vec{F}_L(\boldsymbol{A},\boldsymbol{N}_C) &= ([\,0.75 \quad 0.25\,]\circ[\,1 \quad 3\,])\cdot\mathbf{1} \\ \vec{F}_L(\boldsymbol{A},\boldsymbol{N}_C) &= [\,0.75 \quad 0.75\,]\cdot\mathbf{1} \\ \therefore\quad G_L(\vec{F}_L(\boldsymbol{A},\boldsymbol{N}_C)) &= 0.75 \end{aligned} \tag{3.15}$$

As can be seen with lower latency value achieved in the relaxed case, by allowing the task to be split over multiple platforms, a better metric value can be achieved by using computational resources cooperatively as opposed to exclusively.

However, it is possible that a multiple platform allocation might result in worse performance for another metric, as discussed in Section 2.3.2, for example the energy consumed for the computation, as now two platforms are required instead of only one. I address this concern in the next subsection.

3.3.3. Multimetric Pareto Spaces

As the metrics under consideration are also known *a priori*, additional constraints may be added to the optimisation program for every other metric being considered besides k, in this case n, as given in (3.16). This program requires that the allocation also satisfies all of the metric values specified in addition to optimising M_k.

$$\begin{aligned} \underset{\boldsymbol{A} \in \mathbb{R}_{+}^{\mu \times \tau}}{\text{optimise}} \quad & G_k(\vec{F}_k(\boldsymbol{A}, \boldsymbol{V_p})) \quad \boldsymbol{V_p} \in \mathbb{R}^{\mu \times \tau \times v}, \\ \text{subject to} \quad & \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1, 2, \ldots, \tau, \\ & G_n(\vec{F}_n(\boldsymbol{A}, \boldsymbol{V_p})) = M_n \quad n = [1, 2, \ldots, m] - k. \end{aligned} \tag{3.16}$$

Multiple instances of the multimetric optimisation program given in (3.16) can be used to generate the domain Pareto surface, representing the combination of the heterogeneous computing resources in terms of the domain metrics. For this surface to be populated, a range of metric values are required for all metrics that satisfy the program. This ranges of metrics can be found using the ϵ-constraint method, as described by Kirlik and Sayın [72].

3.3.4. Partitionability Criteria

For the feature that the domain specific approach enables optimal partitioning of tasks to proved, a viable means for solving these optimisation problems must be found amongst the approaches described in Section 2.3.3 .

To be viable, a partitioner would firstly need to produce an allocation that obeys all of the constraints, and secondly do so for a cost, whether in latency or in terms of resources, that is less than that of a substantial task workload. Similar to the extrapolation criteria for domain metric models, this criteria is heuristic.

To be optimal, a partitioner should produce an allocation that is Pareto optimal, i.e. any change to the allocation will result in a worse value for at least one of the metrics

3.4. Methodology Conclusion

In this chapter, I have described a domain specific methodology for heterogeneous computing for implementations with three features: portability, predictability and partitionability. I have described and illustrated these features using examples from the domain of linear algebra arithmetic and image filtering, whilst drawing upon the literature, as summarised in Section 2.4.

In my description of the features I have motivated why the domain specificity of the task description enables the features, as well laid out criteria by which the existence of the these features may be assessed.

In the next Part of this dissertation, I evaluate the proposed methodology using a case study from the domain of computational finance, using the criteria laid out in Sections 3.1.4, 3.2.4 and 3.3.4.

Part II.

The Case Study: Derivatives Pricing

4. The Derivatives Pricing Domain

In first part of this dissertation I have provided the background to my thesis that domain specificity can provide the three features of portability, prediction and partitioning for heterogeneous computing systems. In the Literature Review, I described the current state of the art in heterogeneous, domain specific and distributed computing, and how these fields inform this thesis. Building upon the observations in the Literature Review, I proposed a methodology in Chapter 3, formulating a domain specific approach to heterogeneous computing, so as to achieve the three features. In this Part, I apply my proposed methodology to a detailed case study so as to demonstrate it in practice, as well as evaluating its validity.

However, before considering the three features of my methodology in practice, in this chapter I define the domain of derivatives pricing, in the larger area of computational finance that I will use as an example application domain in my case study. I motivate and describe the background to the derivatives pricing domain. I then provide the domain's semantic model, as described in Section 2.2.2, describing its *types* and *functions*, that will be used as inputs into the methodology described in Chapter 3.

4.1. Computational Finance Background

Complex financial products such as derivatives are widely used in modern commerce, accounting for more than \$63 trillion of active financial products today [73, 74]. Derivatives allow for sources of uncertainty to be quantified and accounted for as *risk*, helping to ease the movement of capital throughout the globalised economy. These instruments do however pose technical challenges, including the computationally intensive task of finding a value for these risk management vehicles [74]. Option contracts in particular offer a considerable challenge, making a degree of intuitive sense while evading elegant mathematical description.

Over the last eight years the disconnect between how these products' risk is quantitatively evaluated, and the uncertainty the products are meant to reflect has been widely held to be a driver of the Global Financial Crisis of 2008 [75]. In response to the growing consensus on this disconnection, increasingly regulators[1] require that these products are valued in a more coherent and systematic manner. Financial engineers, the domain programmers who undertake this pricing, are typically highly knowledgeable of the intricacies of their domain, but usually not in the computational implementation thereof.

Hence, derivatives pricing is an application domain that has need of heterogeneous HPC but whose domain programmers don't necessarily have the knowledge to make use of it. Hence, I have chosen to use it as a representative case study while answering the dissertation's research questions. This section provides the theoretical background to derivatives pricing tasks and

[1] and common sense

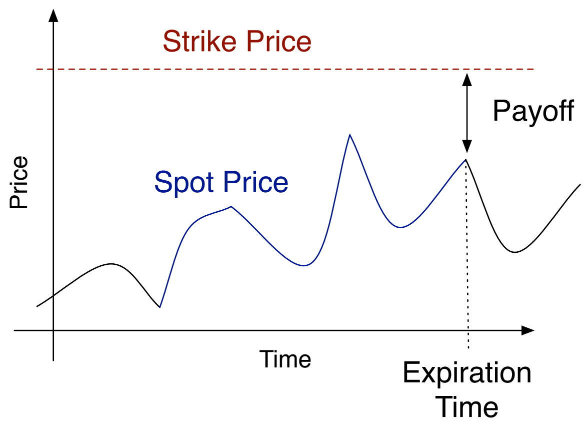


Figure 4.1.: Diagrammatic overview of option valuation, showing the relationship between the underlying asset's spot price, that varies over time, with the option's defined strike price. The difference between the strike price and the spot price, the payoff, gives the option its value.

the popular Monte Carlo pricing algorithm. In describing the background, I will demonstrate the computational intensity of these problems, and hence motivate the requirement of powerful computing resources.

4.1.1. Forward Looking Options

Option contracts are agreements where a *holder* pays a *premium* to the contract *writer* in order to obtain a set of rights with regards to an *underlying asset* , such as 100 shares of a certain stock, foreign currency or commodity. These rights allow the *holder* to either buy or sell the underlying asset at a defined *strike price* under defined conditions. The key word in this description is right - the *holder* has bought the right to exercise the option contract if they so choose, and is in no way obligated. It is assumed that they will only do so when it is to their benefit, i.e. the size of the difference between the strike and spot prices or *payoff* is positive [74]. An overview of the relationship between these key components is given in Figure 4.1.

Options are hence a type of derivative, as the *value* of the option is derived from the price of the underlying asset at a certain *exercise time* in the future. Products with a single exercise time are popularly known as European Options, due to the geographic origin of the type of the financial product. The *holder* has paid a *premium* in order to adopt a position in future on the value of an asset. The *writer* has received the *premium* in order to assume the risk that the future position adopted by the *holder* will turn out to be advantageous [74].

Forward-looking Options are those options which only have value at a single point in the future (the exercise point), which must be considered when calculating its value.

4.1.2. Common Forward Looking Options

Table 4.1 provides a summary of the behaviours of the option types considered in this dissertation, as described below [74]. The forms considered are the call versions of the options, where the option holder is paying for the right to *short* or sell the underlying asset at the exercise time.

Table 4.1.: Overview of call versions of common option pricing problems. The barrier options are knock-out barrier option. Defined problem parameters are the spot price (S_t) at time t, the strike price (K), the Barrier Value (H). Defined problem variables are barrier crossed (B_t) at time t, culmulative sum (C_t) at time t.

Option Type	Lifetime	Payoff
European	-	$V_E = \max(K - S_T, 0)$
Barrier	$B_t = \begin{cases} 1 : S_t > H, B_{t-1} = 1 \\ 0 : S_t < H \end{cases}$	$V_B = \begin{cases} V_E : B_T = 0 \\ 0 : B_T = 1 \end{cases}$
Double Barrier	$B_t = \begin{cases} 1 : H_L > S_t, S_t > H_U, B_{t-1} = 1 \\ 0 : H_L < S_t < H \end{cases}$	$V_{DB} = \begin{cases} V_E : B_T = 0 \\ 0 : B_T = 1 \end{cases}$
Double Digital Barrier	$B_t = \begin{cases} 1 : H_L > S_t, S_t > H_U, B_{t-1} = 1 \\ 0 : H_L < S_t < H \end{cases}$	$V_{DDB} = \begin{cases} 1 : V_{DB} > 0 \\ 0 : V_{DB} \leq 0 \end{cases}$
Asian	$C_t = C_{t-1} + S_t$	$V_A = \max(K - \frac{C_T}{T}, 0)$

In addition to the European or Vanilla options described, a variety of "exotic" options are defined: Barrier, Binary and Asian options. These derivative contract forms have arisen in practice, for a variety of reasons, usually to mitigate and control various types of risks, such as price shocks in the case of barrier options or market manipulation in the Asian option case.

European options

European options are the original form of option contracts. A single, constant strike price (K) is defined in relation to the asset price(S) for the option at its initiation and a single date is set as the exercise point (T). In a risk neutral world, i.e. which ascribes no value to risk, the value for a put and call option at time t are respectively defined in (4.1) and (4.2).

$$V_{c,t} = \max(e^{-r(T-t)}(K - S_T), 0) \tag{4.1}$$

$$V_{p,t} = \max(e^{-r(T-t)}(S_T - K), 0) \tag{4.2}$$

The non-zero or *In The Money* value of the option is made up of two factors:

1. $e^{-r(T-t)}$ - the *discount factor*, effectively back-dating its value to the current time (t) using the risk free interest rate (r).
2. $(K - S_T)$ in the case of a put option or $(S_T - K)$ in the case of a call option - the *intrinsic value* of the option at the exercise point.

Barrier Options

Barrier options are a form of "exotic" option. Similar to a European Option, it may only be exercised at a certain point in time, however the value of the option is also based upon all of the values that the underlying asset has taken before the exercise point(s). A *out* or *knock-out* barrier option is where the option becomes worthless or invalid if the spot price crosses a certain price barrier (H) during the option's lifetime, as given in Table 4.1. The reverse is true of an *in* or *knock-in* barrier option. Considerable care is taken in defining exactly what constitutes the crossing of the barrier.

Digital options

Digital or binary options are another form of exotic options. In addition to having European exercise properties, rather than having a value that it is based upon the asset's price, a defined payoff is paid out either in the form of cash (P) or the underlying asset if a defined condition is met. This condition may take the form of a strike price being greater than the underlying's spot price, or a certain event occurring such as a market index being above a certain level. A digital option be viewed as a bet, i.e. a prediction with a payoff attached to it.

Similarly to Barrier Options great care is taken in defining the condition upon which the binary option is considered valid or not. In the case of *cash-or-nothing* digital options, the value of the options are contingent on the payout amount and whether the final value is above or below of the strike price, as in Table 4.1.

Asian or Average options

Asian options are also exotic. As with a European Option, it is exercised at a certain point in time. However, instead of the underlying value of the asset at maturity being used to calculate the value, an average of the asset value over a defined period of time is used ($\frac{C_T}{T}$), where C_T is the cumulative sum of the underlying's price at exercise.

4.1.3. Monte Carlo Pricing Algorithm

While the previous subsection described a variety of types of derivatives, in this subsection, I describe a method for attaching a value to these contracts.

The popular Monte Carlo technique for the valuation of options uses random number generators to create simulations or paths of the underlying assets, and uses these simulations to generate the distribution of option values over many scenarios. This distribution is then used to find the average option value. As the number of paths is increased, the price converges on the true option value, according to the asset price model used [74, 34]. An overview of the algorithm is given in Figure 4.2.

The technique is derived from the expression of the option value as the integration of all possible option values (V) with respect to the risk-neutral, probability space ($\mathbb{P}$) defined by the asset model (w), i.e. $\int_w V(w)d\mathbb{P}(w)$. This value must then be back-dated to the present time (t) using the risk-free interest rate (r) and the product's expiration time (T), i.e. $e^{-r(T-t)}$. The Monte Carlo technique can be seen as a technique for discretising the domain of the integration

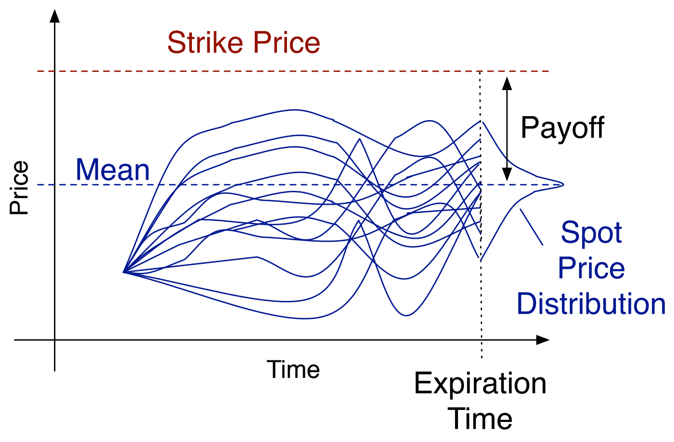


Figure 4.2.: Diagramatic overview of Monte Carlo Option Pricing algorithm. Many simulations or "paths" of the underlying spot price are performed according to a stochastic model of price evolution, with the payoff values at expiration collected. The mean value of the payoff is thus the most probable value of the option, converging upon the true payoff as the number of simulations is increased.

and the probability distribution, and hence finding a numeric approximation, as given in (4.3). However, the algorithm does require a model for simulating the underlying asset's evolution over time so that the multiple values at the expiration time can be found (x_i).

$$V_t = e^{-r(T-t)} \int_w V(w) d\mathbb{P}(w) \approx e^{-r(T-t)} \frac{1}{N} \sum_{i=0}^{N-1} V(x_i) \tag{4.3}$$

Asset Price Evolution Models

To create the underlying paths a model for the asset price evolution is required. One of the most famous models is the Black-Scholes model [74], given in (4.4), which assumes the underlying asset price grows over time (t) at a constant rate (μ), while also fluctuating according a Gaussian Random Process (W_t) scaled by a volatility factor (σ).

$$dS = \mu S dt + \sigma S dW_t \tag{4.4}$$

However, empirical evaluations have shown that the Black-Scholes model doesn't capture commonly observed behaviour of prices in a market. This is particularly with respect to the scale of volatility, which has been shown to vary considerably over time. Hence, alternative models which allow for varying volatility, such as the Heston model [76], are considered.

In the Heston model, both the path's asset price and volatility vary stochastically. In a Heston model-based asset, as given in (4.6), the Variance(V) fluctuates in accordance with the mean rate of revision(κ), long run mean(θ) and volatility of variance(σ) as well as another Gaussian Random Variable (W_t^2), which is related to the first Gaussian Value (W_t^1) in (4.5) by a correlation factor (ρ), as given in (4.7).

$$dS = \mu S dt + \sqrt{V} S dW_t^1 \tag{4.5}$$

$$dV = \kappa(\theta - V)dt + \sigma\sqrt{V}SdW_t^2 \tag{4.6}$$

$$dW_t^1 dW_t^2 = \rho dt \tag{4.7}$$

Discretising the paths

To use a model of asset price evolution, the lifetime of the option must be discretised into a fixed number of steps (D), as given in (4.8).

$$dt = D\delta t \tag{4.8}$$

This discrete price evolution may be re-expressed as a single lognormal variable with a mean of 0 and a variance proportional to the width of the "slice" of the time step, as given in (4.9).

$$S(k\delta t) = S_t e^{\gamma_k} \tag{4.9}$$

In the case of the Black-Scholes model, the value of γ_k is given in (4.10), where ε_i is a sample drawn from the Gaussian random distribution.

$$\gamma_k = \sum_{i=1}^{k}[(\mu - \frac{\sigma^2}{2})\delta t + \sigma\varepsilon_i\sqrt{\delta t}] \tag{4.10}$$

Hence, the value of the option may be derived for a particular sample path, such as in the case of a European call option (as outlined in Table 4.1) using the formula given in (4.11).

$$V(x_i) = S(k\delta t) - K \tag{4.11}$$

Advantages and Disadvantages

The chief advantage of the Monte Carlo pricing algorithm is that it scales well with respect to increasingly complex underlying models: the computational complexity grows linearly with respect to the complexity of the underlying model. This linear growth is in contrast to other approaches, such as the finite difference method, which grow exponentially with each stochastic variable added to the underlying [74]. These quickly become unwieldy if too many dimensions, as required by some underlying models, must be considered.

Another advantage is that it lends itself to parallel execution, being an "Embarrassingly Parallel" algorithm [21, 65] as each path can be simulated in parallel.

The Monte Carlo algorithm is however computationally intensive relative to other derivative pricing methods, due to the generation of the random numbers required at each time step in the path. As many random numbers are required in a computation, hence high quality, computational expensive random number generation procedures have to be used.

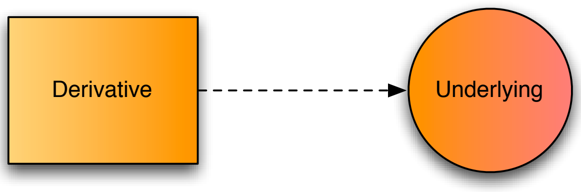


Figure 4.3.: Simplified relationship between derivative and underlying domain concepts

4.2. Derivative Pricing Semantic Model

While the previous section introduced some common types of options and an algorithmic means for pricing options, in this section I will now codify these problems into a semantic model as described in Section 2.2.2.

I will use an empirical definition of application domain, so the data types and functions within the domain should mirror the *nouns* and *verbs* used by domain programmers to describe the operations within the domain. This follows the method proposed by the early advocates of Object Orientation in creating application libraries [30].

I first consider the fundamental types of objects in the derivatives pricing domain, underlying assets and derivative products, and then suggest two valid transformations that can be applied to these types, based upon relationships between these data types. I then describe the domain function, pricing, and illustrate how the Monte Carlo algorithm is applied in performing it. I also identify the execution characteristics of the algorithm, identifying the opportunities for parallel execution.

To illustrate these explanations, I have used the option pricing benchmark from Technische Universität Kaiserslautern[2]. The benchmark is a portfolio of twelve, mostly barrier option pricing problems with six Heston model-based underlyings. The product types within the Kaiserslautern portfolio are a single European option, three barrier options, eight double barrier options and a single digital, double barrier option. The parameters of the pricing problems were chosen carefully to cover a wide variety of scenarios, including "normal" market conditions, as well as periods of high volatility.

As analytic methods for valuing Heston model-based options do not exist [74], part of the benchmark is a 2068 line, GNU Octave reference implementation. This implementation computes the reference values of the options using a Monte Carlo algorithm. The output of the computation is the average option value found, as well as the degree of precision to which that answer has been found.

4.2.1. Domain Data Types: Underlyings and Derivatives

Within this domain there are two fundamental data types, **derivative** products which are being evaluated, and the **underlying** assets from which these derivative derive value. This relationship is illustrated in Figure 4.3, and the options and underlyings that make up the Kaiserslautern benchmark are illustrated in Figure 4.4.

[2]`http://www.uni-kl.de/en/benchmarking/option-pricing/`

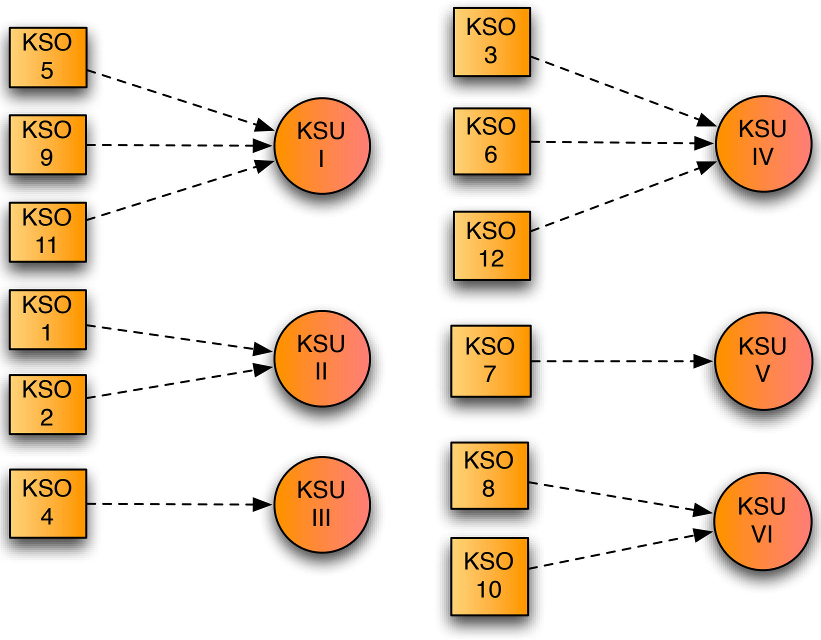


Figure 4.4.: Diagrammatic overview of relationships in the option pricing tasks within Kaiserslautern Option Pricing Benchmark

The **underlying** asset encapsulates the probabilistic model, such as the Black-Scholes or Heston, and its parameters being used to predict the behaviour of the asset under consideration, for example a stock or commodity price. The **derivative** product embodies the details of the option contract both during the lifetime of the option as well at its expiration, as described in the previous section. Hence, in a given pricing problem the derivative provides its underlying asset the relative or delta time that is next required in its valuation. In turn, the underlying asset returns the current price of the asset, as well as the current point in time that the underlying currently represents.

4.2.2. Domain Function: Pricing

Interaction of Underlying Assets and Derivative Products

The pricing function finds the value of the specified derivative product by calling the behaviours of the derivative product as well as the underlying asset that it depends on. The underlying asset has two behaviours - the *path initialisation*, where its initial parameter values are set, as well as the calculations that determines its evolution over the course of its *path* (as given in (4.10)). The derivative product has three behaviours: as well as the *path initialisation* and the *path* behaviour, it also has the *payoff* calculation, for its value at the exercise time. Examples of the path and payoff behaviours for options can be found in Table 4.1.

The interaction between the different behaviours of both concepts in a sequence, as is would be the case in the Monte Carlo algorithm, is captured in Figure 4.5.

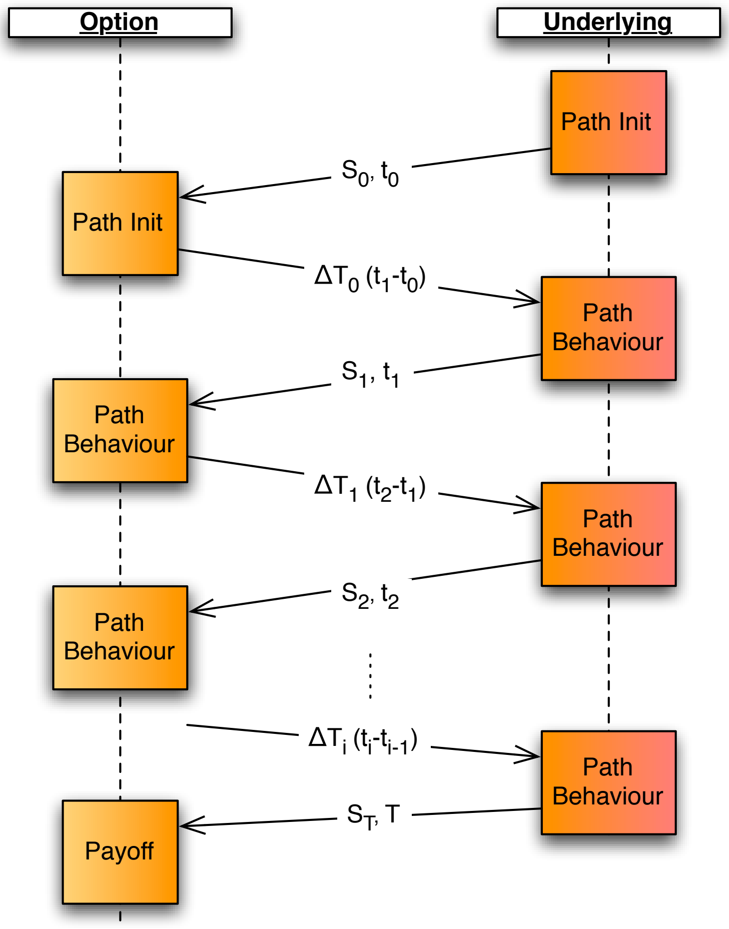


Figure 4.5.: Interaction of Underlying and Option concepts

Listing 4.1: Monte Carlo Option pricing expressed as MapReduce Programming Pattern

```
MAP:
  for(i=0; i<PATHS; ++i){
    state = path_init(seed++);
   PATH:
      for(j=0; j<PATH_POINTS; ++j)
        state = path(state);

    value[i] = payoff(state);
  }

REDUCE:
  for(i=0; i<PATHS; ++i)
    result += value[i]/PATHS;
```

Implementation of Monte Carlo Pricing Algorithm

As described in Section 4.1.3, the pricing function can be performed using the Monte Carlo algorithm. The algorithm finds the price for a derivative by simulating its underlying asset and corresponding exercise value multiple times, and then taking the average value.

Although, as noted above, while the Monte Carlo algorithm is computationally intensive, it lends itself to parallel execution. In fact, it is the canonical “Embarrassingly Parallel” algorithm that fits neatly in the MapReduce programming pattern [21] as demonstrated in Listing 4.1.

The program is comprised of three loops, with the majority of the computational effort within the double loop nest labeled Map. The algorithm requires at least two variables to be defined upon implementation: *PATHS,* the number of simulations required, and *PATH_PATHS*, the number of steps within each simulation.

The inner path loop, which is bound by *PATH_POINTS*, is data dependent, reliant upon the results of the previous operation. Depending upon the number of points in the simulation path, this loop presents an opportunity for pipeline parallelism, where each iteration of the loop could be considered as a potential stage in a pipeline.

The outer, Map loop is bound by *PATHS*, and its iterations are completely independent, and hence can be computed in parallel. By chunking the number of paths evaluated, the calculation can be performed in a task parallel fashion. Furthermore, provided the random number generation procedure used is deterministic, for example a combined Tausworthe uniform random number generator [77] coupled with a Box-Muller transformation [78], the calculation can be computed in a data parallel fashion.

4.3. Derivatives Pricing as a Computational Domain

In this chapter, I have described the background to derivatives pricing in computational finance, and how it can formulated as an application domain. This formulation is with a view towards applying the domain specific methodology described in Chapter 3.

I have defined the domain data types: underlyings and derivatives. Underlyings capture the parameters and evolution of financial assets such as stocks and commodities, while derivatives capture the same for the derivative products such as forward looking options. I also defined the sole domain function, derivative pricing, which is finding the value of the derivative product, based upon the underlying.

In the next three chapters I apply the domain specific methodology to the derivative pricing domain, and use it to evaluate the methodology.

5. Porting Derivatives Pricing

As noted in Section 1.1, increasingly programmers are faced with an ever growing array of computational architectures. These include conventional CPUs with an increasing degree of parallelism, as well as more exotic platforms such as GPUs and FPGAs, which theoretically offer performance too good to ignore. A further development is the combination of these platforms, for example Intel's Xeon Phi architecture, which offers x86 architectural units with the data parallelism of a GPU, or Xilinx's Zynq System-on-Chips, offering both conventional CPU and reconfigurable logic tightly coupled.

The reality is that taking advantage of these heterogeneous computing technologies is a challenge. Even if the considerable orientation and interfacing problems of the platforms have been overcome, there is still the broader, conceptual question that must be answered to make efficient use of the platform's architectural features. The solution I propose, as others have [4], is that the majority of programmers, who often work in a particular application domain, don't address this challenge. Rather system programmers, as described in Section 2.2.1, take advantage of the regular structures and relationships within these domains, as made explicit in application frameworks and DSLs, to enable efficient execution on a wide range of heterogeneous computing platforms.

In describing the efficient, portable implementation, I introduce the Forward Financial Framework (F^3), a heterogeneous computing framework for derivatives pricing that I have been developing since early 2012[1]. The vision of the framework is to enable financial engineers to specify their derivative pricing tasks at a high level, using object orientated constructs that mirror derivatives pricing domain concepts. The framework can then implement the specified pricing tasks efficiently on any of a wide range of heterogeneous computing platforms. Throughout the rest of this case study, I use the framework as a demonstration of the domain specific methodology I have proposed.

In this chapter, I demonstrate how the domain specific methodology I proposed in Chapter 3 enables portable, efficient execution of derivatives pricing on heterogeneous computing platforms. This demonstration has two part: first, I describe the implementation of the derivatives pricing application domain upon a wide range of heterogeneous computing platforms, including multicore CPUs, GPUs and FPGAs. In the second part, I provide an experimental evaluation of the degree to which the criteria for this feature, as described in Section 3.1.4, have been achieved.

[1]The framework is Open Source under GNU Public License, with the full source code available at `https://github.com/Gordonei/ForwardFinancialFramework`

5.1. Implementing Portable Derivatives Pricing

In this section I describe how the semantic model described in Section 4.2 is implemented within F^3, using the domain specific development process described in Section 2.2.2.

The context-free grammar for F^3 can be found in Listing 5.1, described using extended Backus-Naur form. F^3 is an object-orientated application framework [30], implemented at the top-level using the Python programming language. The API documentation for F^3 can be found in Appendix B.

Listing 5.1: The Forward Financial Framework's context-free grammar.

⟨*S*⟩ ::= ⟨*command*⟩

⟨*command*⟩ ::= '`generate`' ⟨*solver*⟩ | '`compile`' ⟨*solver*⟩ | '`execute`' ⟨*solver*⟩

⟨*solver*⟩ ::= ⟨*portfolio*⟩ ⟨*platform*⟩

⟨*platform*⟩ ::= ⟨*hostname*⟩ ⟨*type*⟩

⟨*hostname*⟩ ::= ⟨*string*⟩

⟨*string*⟩ ::= [a-zA-Z0-9]+

⟨*type*⟩ ::= '`POSIX-CPU`' | '`OpenCL-GPU`' | '`OpenCL-FPGA`' | '`OpenSPL-FPGA`' | '`VivadoHLS-FPGA`'

⟨*portfolio*⟩ ::= ⟨*derivative*⟩+

⟨*derivative*⟩ ::= ⟨*underlying*⟩ ⟨*strike-price*⟩ ⟨*derivative-lifetime*⟩ ⟨*call*⟩

⟨*strike-price*⟩ ::= ⟨*positive-real*⟩

⟨*derivative-lifetime*⟩ ::= ⟨*positive-real*⟩

⟨*call*⟩ ::= ⟨*binary*⟩

⟨*binary*⟩ ::= '`FALSE`' | '`TRUE`'

⟨*underlying*⟩ ::= ⟨*rfir*⟩ ⟨*spot-price*⟩

⟨*rfir*⟩ ::= ⟨*real*⟩

⟨*spot-price*⟩ ::= ⟨*positive-real*⟩

⟨*real*⟩ ::= '`-`' ⟨*positive-real*⟩ | ⟨*positive-real*⟩

⟨*positive-real*⟩ ::= ⟨*digit*⟩+ ('`.`' ⟨*digit*⟩+)?

⟨*digit*⟩ ::= '`0`' | '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`'

In this section, I first describe how the domain data types, underlyings and derivatives, and the domain function, pricing, are implemented using Python classes in F^3. I then describe the supporting infrastructure that enables tasks specified using the domain specific abstractions to

Listing 5.2: Example from Kaiserslautern Option Pricing Benchmark of how underlying domain data types are depicted in the Forward Financial Framework.

```
#Kaiserslautern Underlying II
KSU_II = Heston_Underlying(
   rfir = 0.05,
   current_price = 100,
   mean_rate = 0.09,
   vol_vol = 1,
   corr = −0.3,
   rev_rate = 2,
   curr_vol = 0.09)

#Kaiserslautern Underlying III
KSU_III = Heston_Underlying(
   rfir = 0,
   current_price = 100,
   mean_rate = 0.09,
   vol_vol = 1,
   corr = −0.3,
   rev_rate = 1,
   curr_vol = 0.09)
```

be executed on a variety of heterogeneous platforms with portable performance, in accordance with the grammar given in Listing 5.1. Finally, I show how domain knowledge is applied within the framework, making the implementations portably efficient, as outlined in Section 3.1.3.

5.1.1. Derivative Pricing Data Types and Function

F^3 has its core three base classes that mirror key concepts in the financial engineering domain: *Derivatives*, *Underlyings* and *Solvers*. All three can be extended utilising object inheritance as required[2], allowing for broad expression within the confines of the application domain.

Underlying and Option Data Types

Within the derivatives pricing domain I consider, as described in Chapter 4, derivatives and underlyings are the two domain data types. These datatypes are represented as classes that inherit from the base classes of *Option* and *Underlying* respectively. Listings 5.2 and 5.3 give an example of how options and underlyings from the Kaiserslautern option pricing benchmark would be captured, while Figure 5.1 gives a graphical representation of the code.

Underlyings represent those assets from which derivatives derive value. In F^3, all underlyings are represented by classes that correspond to the models that the behaviour of the underlying is based upon, such as the Heston or Black-Scholes. The attributes of the data type class are needed to calculate the asset's value at a given point in time. The underlying class also has modifier methods for changing these attributes. The first, the *path_init*, sets the values of the attribute at the model's inception. The second method, *path*, modifies the attributes in response

[2]Possibly by a local developer, as described in Section 2.2.1

Listing 5.3: Example from Kaiserslautern Option Pricing Benchmark of how option domain data types are depicted in the Forward Financial Framework.

```
#Kaiserslautern Option #1
KSO_1 = European_Option (
   underlying = [Heston_II],
   call = True,
   strike_price = 100,
   time_period = 5)

#Kaiserslautern Option #2
KSO_2 = Barrier_Option (
   underlying = [Heston_II],
   call = True,
   strike_price = 100,
   time_period = 5,
   points = 4096,
   out = True,
   barrier = 120)

#Kaiserslautern Option #4
KSO_4 = Double_Barrier_Option (
   underlying = [Heston_III],
   call = True,
   strike_price = 100,
   time_period = 5,
   points = 4096,
   out = True,
   barrier = 120)
```

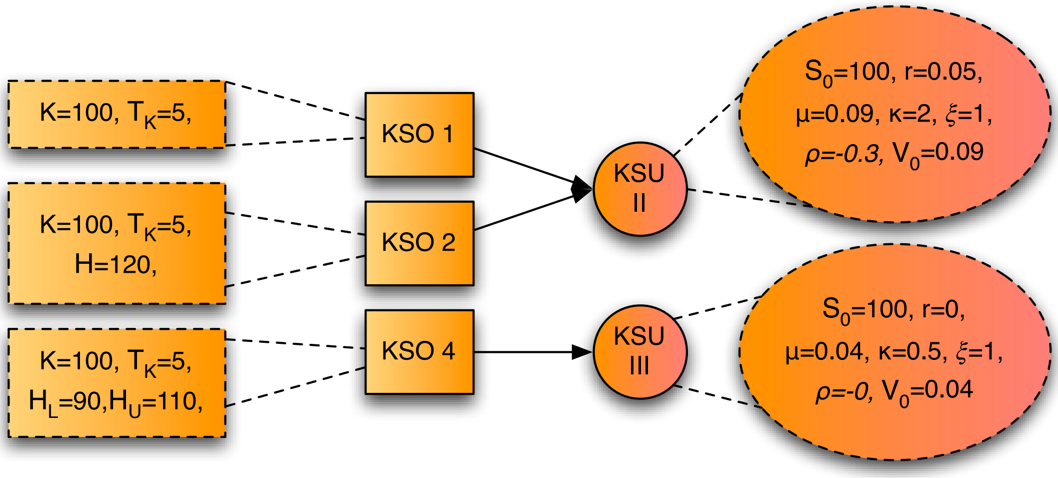

K=100, T_K=5,
K=100, T_K=5, H=120,
K=100, T_K=5, H_L=90,H_U=110,
KSO 1
KSO 2
KSO 4
KSU II
KSU III
S_0=100, r=0.05, μ=0.09, κ=2, ξ=1, ρ=-0.3, V_0=0.09
S_0=100, r=0, μ=0.04, κ=0.5, ξ=1, ρ=-0, V_0=0.04

Figure 5.1.: Example of a subset of options from the Kaiserslautern Benchmark rendered by the domain programmer within the Forward Financial Framework. The diagram depicts the relationship between the underlying domain datatypes defined in Listing 5.2 and the options defined in Listing 5.3.

to changes in time, in accordance with the defined model of price evolution [76, 74]. In order to provide the *path* behaviour, the underlying object has to rely on a data source to determine the evolution of its variables, such as a random number generator or historical data.

The efficient valuation of *Derivatives* is the main goal of the domain. Similar to the underlying, the derivative data type class aggregates behaviours and attributes required to calculate a value for the derivative under specified conditions. However, unlike underlyings, there is a further *payoff* modifier method defined, which is used to set the resulting value attribute of the derivative, given the final underlying price(s) at the derivative's defined exercise point as well as the derivative's attributes.

A limitation of F^3 is that currently only derivatives with European exercise properties, i.e. that have only a single exercise point in the future, are supported.

Derivative Pricing Function

While Derivatives and underlyings are the fundamental types of F^3, the Solver class provides the conceptual container for their interaction. Solvers contain both underlying and derivative objects, but also describe the nature of the calculation being performed, capturing the interaction between the derivative and underlying objects.

A solver within F^3 has three behavioural descriptions: *generate*, *compile* and *execute*: As the name suggests, *generate* uses the specified platform to generate the required code in order for it to be compiled and executed upon its target platform. It is at this point that the framework makes use of advanced object-oriented features such as introspection in order to identify explicit relationships between underlying and derivative objects. *Compile* and *execute*, interact with the specified platform to compile and execute the specified pricing task, returning the result to the end user in the high level means of expression.

A further limitation of F^3 is that only the Monte Carlo algorithm for implementing option pricing is currently supported.

5.1.2. Supporting Portable Execution

Listing 5.4 and Figure 5.2 provide an overview of the flow from problem definition to implementation upon two different platforms. In this example, the portfolio of derivative products and underlying assets defined in Figure 5.1 are partitioned into two groups and applied to two different solvers, which are then executed on different platforms.

Below I explain the flow in terms of the commands allowed in the grammar given in Listing 5.1, i.e. from generate to execute, as illustrated in Figures 5.1 and 5.2. This flow fulfils the supporting infrastructure requirements for the domain specific methodology, as described in Section 3.1.2.

Generate

Domain Specific Task Description: The financial domain programmer must specify the derivative products they wish to value, and the underlying asset models upon which the derivatives depend, as instances of appropriate classes within a Python script, as illustrated in Listing 5.2 and 5.3.

Listing 5.4: Forward Financial Framework code for the implementation of the options described in 5.3

```
#Creating the platforms and solver objects
posix_cpu = POSIX_CPU(
   hostname = "localhost")
opencl_fpga = OpenCL_FPGA(
   hostname = "localhost")

mc_solver_cpu = MonteCarlo(
   options = [KSO_1,KSO_2],
   paths = 1e7,
   platform = posix_cpu)
mc_solver_fpga = MonteCarlo(
   options = [KSO_4],
   paths = 1e7,
   platform = opencl_fpga)

#Calling the CPU solver to generate, compile and execute
mc_solver_cpu.generate()
mc_solver_cpu.compile()
mc_solver_cpu.execute()

#Calling the FPGA solver to generate, compile and execute
mc_solver_fpga.generate()
mc_solver_fpga.compile()
mc_solver_fpga.execute()
```

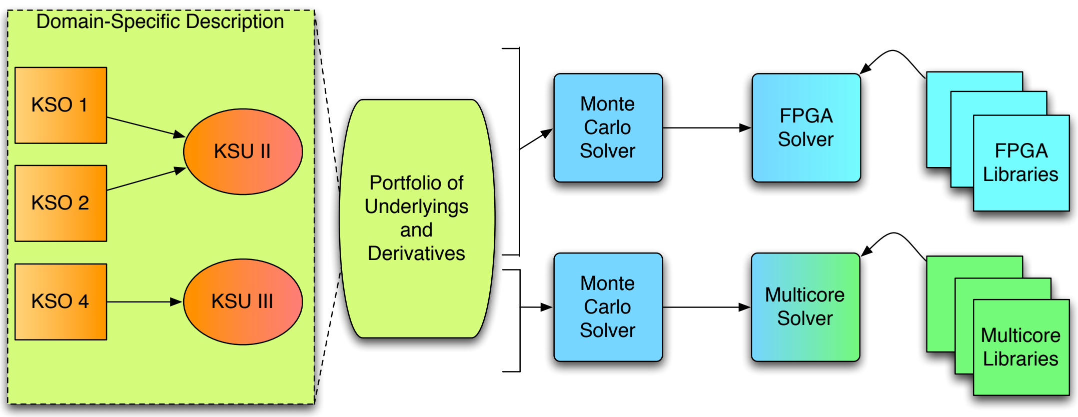


Figure 5.2.: Overview of Forward Financial Framework flow from problem specification by the domain programmer to implementation on the target heterogeneous system.

The programmer then groups together these derivatives into portfolios, and allocates the portfolios to instances of the solver class. A platform class instance is also specified for each solver, which represents the computational platform that the portfolio will be evaluated upon, as is illustrated for a multicore CPU and FPGA, as illustrated in Listing 5.4. This solver instance, containing both the portfolio and platform, may then be used to generate the required code.

Platform solver code generation: The code required to value the portfolio of derivatives can then be generated for the target platform by F^3 by calling the generate behaviour of the solver instance.

The code generated implements the algorithm associated with the solver's class. Currently only a Monte Carlo-based solver has been implemented.

The solver class is able to generate the required code correctly for a wide array of financial products and asset models for two reasons: firstly, the general structure of the pricing algorithm is known in advance, and secondly, the Python programming language supports Introspection, the ability to examine the structure of code at run-time. Knowing the structure of the algorithm in advance means that the solver class can contain a template for the algorithm upon the target platform, similar to Listing 4.1, which can then be populated with the behaviours for the specified portfolio of products and underlying models using introspection.

An example of the OpenCL kernel code generated by F^3 for the 4th Kaiserslautern option, a Double Barrier option with a Heston-based underlying, is given in Listing 5.5.

Supporting code: An important clarification is that only the solver algorithmic code for a particular platform is generated by the framework - supporting libraries written in the targeted platform's code are also required.

Two types of supporting libraries are required: The first type is a description or translation of the underlying and derivative classes into a form which can then be implemented on the target platform. For example, in the case of the multicore CPU implementation, C code versions of the underlying and derivative objects are required that implement the product or asset's specified behaviours. The second type is the general utility libraries required to implement the solver algorithm upon the desired platform, for example code for generating Gaussian random numbers.

An example of the supporting C code for European options, which is used by both the multicore CPU and OpenCL solver generated-code, is given in Listing 5.6.

My experience has been that that creating this supporting code has been by far the least intensive aspect of extending the framework to a new computing platform. Rather, I found writing the solver platform code takes significant development time, hence my decision to automate its implementation.

Furthermore, I believe writing this derivative and underlying platform code would be well within the grasp of the determined financial domain programmer, such as the local developers described in Section 2.2.1, given the examples already provided within the framework.

Listing 5.5: Example of solver code generated by the Forward Financial Framework: OpenCL Pricing kernel for Kaiserslautern Option 4

```
kernel void mc_solver_opencl_gpu_he_1_do_1_kernel(
  const uint path_points,
  const uint seed,
  const uint chunk_size,
  const uint chunk_number,
  const heston_attributes u_a_0,
  const double_barrier_attributes o_a_0,
  global FP_t *value_0){
  //getting unique ID
  int i = get_global_id(0);

  //reading parameters from host
  uint temp_path_points = path_points;
  uint temp_chunk_size = chunk_size;
  uint temp_chunk = chunk_number;
  uint temp_seed = seed;

  //copying parameters from host
  heston_attributes temp_u_a_0 = u_a_0;
  double_barrier_attributes temp_o_a_0 = o_a_0;

  //creating kernel variables
  heston_variables temp_u_v_0;
  uint seed_offset = i*KERNEL_LOOPS+temp_chunk_size*temp_chunk;
  FP_t spot_price_0,time_0;
  double_barrier_variables temp_o_v_0;
  FP_t temp_value_0 = 0.0;
  FP_t temp_value_sqrd_0 = 0.0;

  seed(local_seed + 1 * seed_offset,&(temp_u_v_0.state));
  for(int k=0; k<KERNEL_LOOPS; ++k){
   //initiating the path and creating path variables
   heston_path_init(&temp_u_v_0,&temp_u_a_0);
   spot_price_0 = temp_u_a_0.current_price*exp(temp_u_v_0.gamma);
   time_0 = temp_u_v_0.time;
   double_barrier_path_init(&temp_o_v_0,&temp_o_a_0);

   //running the path
   for(int j=0;j<local_path_points;++j){
    double_barrier_path(spot_price_0,time_0,&temp_o_v_0,&temp_o_a_0);
    heston_path(temp_o_v_0.delta_time,&temp_u_v_0,&temp_u_a_0);
    spot_price_0 = temp_u_a_0.current_price*exp(temp_u_v_0.gamma);
    time_0 = temp_u_v_0.time;
   }

   //calculating payoff(s)
   double_barrier_payoff(spot_price_0,&temp_o_v_0,&temp_o_a_0);
   temp_value_0 += temp_o_v_0.value;
  }

  //copying result to global memory
  value_0[i] = temp_value_0;
}
```

Listing 5.6: Example of supporting code for the Forward Financial Framework: C code for European Options, used by both multicore CPU and OpenCL implementations

```
//Option initialisation function
void european_init(
  FP_t t,
  char c,
  FP_t k,
  european_option_attributes* o_a){
  option_init(t,c,k,&(o_a->option));
  o_a->time_period = (o_a->option).time_period;
  o_a->strike_price = (o_a->option).strike_price;
  o_a->call = (o_a->option).call;
}

//Option simulation path initialisation function
void european_path_init(
  european_variables* o_v,
  european_attributes* o_a){
  option_path_init(&(o_v->option),&(o_a->option));
  o_v->value=(o_v->option).value;
  o_v->delta_time=(o_v->option).delta_time;
}

//Option simulation path evolution function
void european_path(
  FP_t price,
  FP_t time,
  european_option_variables* o_v,
  european_option_attributes* o_a){
  option_path(price,time,&(o_v->option),&(o_a->option));
}

//Option simulation path payoff function
void european_option_derivative_payoff(
  FP_t end_price,
  european_option_variables* o_v,
  european_option_attributes* o_a){
  if(((o_a->call) && (end_price < o_a->strike_price))
  || ((o_a->call) && (end_price > o_a->strike_price)))
   option_payoff(o_a->strike_price,&(o_v->option),&(o_a->option));
  else
   option_payoff(end_price,&(o_v->option),&(o_a->option));
  o_v->value = (o_v->option).value;
}
```

Compile

Once the required platform-specific solver code has been generated, the platform's pre-existing compilation tools can then used to compile the code. The solver instance is capable of managing this process automatically, initiating it when its compile method is called, as in Listing 5.4.

A variety of compilers are used, depending upon the platform: GCC is used for the multicore CPU platforms; Maxeler's MaxCompiler, the Altera OpenCL SDK and Xilinx Vivado HLS is used for FPGA platforms; vendor provided OpenCL SDKs are used for the GPU and remaining coprocessor platforms, such as Intel's Xeon Phi. The solver either calls these tools directly, using Python's built-in Subprocess module, or by interfacing with build systems, such as GNU Make.

Execute

The programmer can then start the execution, i.e. the actual pricing of the specified portfolio on the target platform, by calling the solver instance's execute method, as in Listing 5.4. Similar to compilation, the solver manages this process, using Python's subprocess module to call the compiled platform-specific solver with the appropriate arguments. Execution over Internet Protocol-capable network connections is also supported using the Secure Shell (SSH) protocol, although this requires a compiled solver to be available on the remote host that the solver is being executed upon. Once completed, the execute method returns the value of the derivatives in the portfolio as well as various performance metrics within the Python programming environment.

5.1.3. Enabling efficient execution with domain knowledge

In this subsection I describe how F^3 produces implementations that are efficient by exploiting the potential for parallel execution in the domain specific task structure. As outlined in Section 3.1.3, the domain specific methodology makes the structure of computational tasks explicit at compilation. F^3 shows this in practice, using the structure of the Monte Carlo pricing as described in Section 4.2.2, to enable parallel execution across all of the platforms targeted by the framework.

Below, I describe how this is done in terms of task, data and pipeline parallelism, firstly in general, and then with reference to the different heterogeneous platforms supported.

Task Parallelism

As noted in Section 4.2.2, there is ample opportunity for parallel execution in the Monte Carlo pricing algorithm that all F^3 solvers currently use. To expose this parallelism, I have re-expressed the algorithm, as described in Listing 4.1, in Listing 5.7 by introducing a third, outer loop bounded by P, labeled CHUNK. The iterations of CHUNK can can be computed in parallel, independent from each other.

The domain knowledge that is being exploited is both the lack of dependencies between the iterations in the MAP loop, as well as knowledge of the composition of the algorithm. As the MAP loop accounts for almost all of the computation, this is an "embarrassingly parallel" problem, and hence justifies the use of parallel execution in all but the smallest of problem sizes.

Listing 5.7: Making the Potential Task and Data Parallelism explicit

```
CHUNK:
  for(p=0; p < P; ++p){
    MAP:
      for(i=0; i < PATHS/P; ++i){
        state = path_init(seed++);
        PATH:
          for(j=0; j < PATH_POINTS; ++j)
            state = path(state);
        offset = p * PATHS/P;
        value[offset + i] = payoff(state);
      }
  }
```

F^3 automatically generates task parallel implementations for all of the platforms targeted.

The Pthreads library is used to execute in a task parallel fashion on multicore CPUs. A thread is spawned for each processor core, as reported by the platform's operating system, and the total number of Monte Carlo paths are divided evenly across the threads.

Similarly, for OpenCL, multiple work-groups are used, with the paths shared evenly amongst the groups. The number of work-groups is set by F^3 at run-time according to the number of compute units available on the target device, as reported by the OpenCL API. A further optimisation based upon task parallelism is that the time spent communicating with the platform is "hidden" by further grouping the number of paths into batches. After a batch of paths has been computed, the next batch is started while the completed batch's results are communicated back to the host system. This communication latency hiding is an example of task parallelism,

In the FPGA implementations, multiple instances of the MAP code are used to compute paths in a task parallel fashion. In OpenSPL an architectural loop is used to create multiple instances of the MAP, while the Altera OpenCL SDK allows F^3 to set the number of compute units explicitly. Finally, in VivadoHLS multiple instances are created by calling non-communicating MAP function multiple times at the same scope.

Data Parallelism

Similar to the task parallelism optimisation, a data parallel approach exploits the lack of dependencies between iterations of the MAP loop in Listing 4.1 to compute iterations in parallel. However, data parallel execution additionally requires that the iterations of the loop operate in close to lockstep, with limited or no divergence between the control flow in iterations being computed concurrently. Any divergence results in pipeline stalls, with all the paths being computed in parallel running taking as long as the longest running path.

F^3 enables data parallel execution for OpenCL platforms, as platforms using the programming standard, particularly GPUs, often have considerable data parallel computational capability. Firstly, using the OpenCL API at run-time, the framework sets the number of work-items in each work-group according to the number of processing elements available for the platform. Furthermore, the code used is completely deterministic, with only balanced conditional expressions

Listing 5.8: C-Slow Transformation

```
for(j=0; j < PATHS/C; ++j){
  for (k=0; k < C; ++k)
    state[k] = path_init(seed++);

  PATH:
    for(i=0; i < PATH_POINTS; ++i){
      MAP:
        for (k=0; k < C; ++k)
          state[k] = path(state[k]);
    }

  for (k=0; k < C; ++k){
    offset = j * C;
    value[offset + k] = payoff(state[k]);
  }
}
```

used. This deterministic code ensures that the work-items in a work-group can be executed in a data parallel fashion without additional instructions to manage divergence.

Pipeline Parallelism

To illustrate the potential for pipeline parallelism in Monte Carlo pricing, I use the concept of C-Slowing. While C-slowing is originally a technique in digital circuit design, hiding the latency of operations using memory resource is transferable to many architectures, particularly FPGAs. In Monte Carlo pricing C-slowing is achieved by inverting the loops labeled PATH and MAP while providing an appropriate memory array of size C to maintain the state between iterations of PATH. Unlike task and data parallelism, this allows for parallel evaluation of iterations of the PATH as opposed to MAP loop. I have illustrated pipeline parallelism in Listing 5.8.

The domain knowledge that is being applied here is orthogonal to the task and data parallelism of the previous subsections. The task and data parallel execution is based upon the lack of dependencies between the MAP loop iterations, whereas here I'm exploiting the compile time knowledge of the length of the PATH loop to keep all of the computational resources busy.

F^3 enables pipeline parallel execution in the FPGA implementations. FPGAs are well suited to exploiting pipeline parallelism, as additional logic and memory resources on the device can be used to extend the length of the execution pipeline. The extended pipelines with memory buffers allow for fine-grained parallel execution of many stages of the pipeline, resulting in improved throughput.

The Altera OpenCL SDK already uses pipeline parallelism by default, inserting pipeline buffers so that work-items can be streamed through a pipeline based upon the OpenCL kernel. However, F^3 extends this pipeline parallelism by inserting code pragmas that unroll the PATH loop. Doing so creates a longer pipeline, and hence improves the throughput of the design.

MaxCompiler also pipelines designs automatically, however F^3 adds a C-Slowing transformation, as depicted in Listing 5.8, so as to hide the latency of pipeline stages using memory resources.

Finally VivadoHLS does not pipeline designs by default, and hence F^3 applies pragmas to pipeline designs and unroll the PATH loop so to improve the throughput of the design.

5.2. Evaluation

This section describes my evaluation of the efficiency of F^3's portable implementations upon multiple heterogeneous platform. This evaluation is focused on efficiency, as implicit in evaluating the efficiency of F^3's implementations across a diverse set of heterogeneous platforms, is evidence that the approach is portable to those platforms

I first describe the experimental platforms and latency metric used in these experiments. I then provide a description and the results of the first experiment, which measures F^3's heterogeneous implementations in terms of parallel scaling, a purely intra-platform, relativistic measurement. I then compare the implementations for the different platform, as well as external, programmer implementations of the same problems.

The aim of the first experiment is to assess whether the implementations created automatically by F^3 can make use of the parallelism exposed by prior analysis of the pricing domain function. The second experiment's aim is to then assess whether the benefit provided from doing so makes full use of the platform, and hence are comparable to those created by platform programming experts.

5.2.1. Experimental Platforms and Latency Metric

Platforms

Table 5.1 provides the designations for the heterogeneous platforms used in these experiments. For more information, Tables A.1, A.3 and A.5 provide an overview of the experimental platforms, while Tables A.2, A.4 and A.6 provide more detailed computational characteristics of the different platforms, such as the clock rate and parallel resources.

The multicore CPU platforms span a spectrum from those found in desktop systems to high-end servers. Desktop CPUs, such as the Intel Core i7-2600 are similar to the one that domain programmers might have on their desks. These provide a modest degree of task parallelism at a high clock rate. The server grade CPU, the Intel Xeon E5-2680v2, is easily accessible, thanks to IaaS providers such as Amazon Web Services (AWS). Although at a lower clock rate to the desktop system, it has twice the parallel compute resource as well as considerably larger caches. The final CPU platform, the "manycore" server system, is of the type likely to be shared between many users in a large organisation. The manycore system is comprised of four server grade CPUs, AMD Opteron 6272, with 16 processing cores upon each CPU. In all of the cases, two cores share a floating-point computational unit.

The GPU platforms represent two classes of this platform type. The first class is the workstation grade GPUs, the AMD Firepro W5000 and the NVIDIA Quadro K4000. Boasting

Table 5.1.: Experimental platforms used in proving portability property

Type	Designation	Platform Name	Standard (Tool)
CPUs	Desktop	Intel Core i7-2600	POSIX(GCC)
	Server	Intel Xeon E5-2680v2	POSIX (GCC)
	Manycore	4 x AMD Opteron 6272	POSIX (GCC)
GPUs	NVIDIA Workstation	NVIDIA Quadro K4000	OpenCL (NVIDIA OpenCL SDK)
	NVIDIA Cloud	NVIDIA GK104	OpenCL (NVIDIA OpenCL SDK)
	AMD Workstation	AMD Firepro W5000	OpenCL (AMDAPP)
	Phi	Intel Xeon Phi 3120P	OpenCL (Intel SDK for OpenCL)
FPGAs	ZC706	Xilinx ZC706 1.1	POSIX (Vivado HLS)
	PCIe-A7	Nallatech P385-A7	OpenCL (Altera OpenCL SDK)
	PCIe-D5	Nallatech P385-D5	OpenCL (Altera OpenCL SDK)
	Max3	Maxeler Max 3424A	OpenSPL (MaxCompiler)
	Max4	Maxeler Max 4	OpenSPL (MaxCompiler)

considerable parallel compute resources, these platforms are targeted at graphics, and hence data parallel workloads such as Computer Aided Design (CAD) and digital film rendering.

The second class are massively parallel compute platforms, often called General Purpose GPU (GPGPUs), the NVIDIA GK104 and the Intel Xeon Phi 3120P. The Xeon Phi is most accurately described as a hybrid architecture, being somewhere between a GPU and CPU. Although with fewer parallel compute resources than a high end GPUs, the coprocessor has considerably more sophisticated control logic, implementing many x86 cores upon a single chip.

Finally, the FPGA platforms are largely in a host-CPU, PCIe card-based stand-alone processing unit configuration, similar to GPUs, with the exception of the Xilinx ZC706 platform, which is a System-on-Chip platform, with a processor sharing a silicon die with the reconfigurable fabric [79]. The computational device on the ZC706 is a Xilinx Zynq 7045, an ARM processor that shares the same silicon with reconfigurable logic elements. While generally targeted towards embedded applications, such a platform presents an opportunity to exploit the tight coupling between host CPU and accelerators implemented in the reconfigurable fabric.

Latency Metric

The metric used in both experiments is the option pricing task latency[3]. I interpret latency as wall-clock time, i.e. how long does the platform under consideration take to perform the calculation and return the result to the domain programmer using F^3, as a measurement[4] of absolute time using an external timing reference. This is to ensure that all system overheads such IO and control structures are incorporated in the evaluation. Further details are given in Section A.3.1.

[3]As currently I'm only considering single task workloads, this is equivilant to the makespan

[4]The host CPU's system clock

Table 5.2.: Overview of option pricing tasks used in characterisation

Source	Designation	Option Task Designation
KS[81]	KSO1	H-E
	KSO2, KSO3, KSO11	H-B
	KSO4 - KSO10	H-DB
	KSO12	H-DDB
IC[80]	IC	BS-A

5.2.2. Intra-Platform Performance Characterisation

The first experiment I undertook in proving the efficient portability property was to assess the degree to which the framework takes advantage of parallel computing resources. In the description below I explain how this was achieved in all of the platform implementations. I then make a projection with regards to the ideal parallel scaling of the different platforms. I finish the experiment by providing and discussing the results.

Description

Tasks: I evaluated the latency scaling characteristic using all 12 of the Kaiserslautern benchmark options as well as the Black-Scholes-based Asian option used in Imperial College London's work [80], as given in Table 5.2. The full breakdown of the number of floating point computational operations per option pricing task type may be found in Table A.8.

I used 10 million simulation paths per option pricing task, with 4096 discretisation points within each simulation path.

Latency Acceleration Measurement: Latency acceleration is the latency of a baseline implementation latency divided by the measured latencies for that platform's implementation. I compare the scaling trend against an ideal, linear parallel scaling relationship in which acceleration is equal to the degree of parallelism.

Experimental Procedure: In order to evaluate the strong parallel scaling property, I varied the degree of parallelism, or parallelism factor, in the implementations across the experimental heterogeneous platforms in Table 5.1. The method by which I was able to vary the degree of parallelism across the platforms was determined by the programming standard used.

For the **multicore CPUs** implementations, I varied the parallelism factor by setting the number of POSIX threads (Pthreads) between which the Monte Carlo simulations are evenly divided. The default behaviour of F^3 is to set the number of Pthreads equal to the number of cores available.

On the **GPUs**, the number of work groups dispatched to the OpenCL subsystem was varied in accordance with the specified parallelism factor. Ordinarily, F^3 sets the number of work groups automatically, querying the characteristics of the OpenCL device at compile time, and then sets the number of work groups to a small multiple of the number of compute units. This oversubscription provides the platform scheduler with sufficient scope to overlap memory accesses within each compute unit.

In the **FPGA** implementations, all three standards allow for the number of accelerator instances implemented in the reconfigurable fabric to be varied, and hence executed in parallel. For the Altera OpenCL platforms, this is achieved by unrolling the data dependent PATH loop, whilst in the Max3, Max4 and ZC706 cases this was done by replicating the accelerator in the control logic multiple times. Currently the framework uses a heuristic of the number of parallel instances that can be supported upon the specific chip, in terms of the number of underlying products implemented. I have found that the underlyings, the random number generators therein in particular, account for the majority of the resources used.

Projection

As the Monte Carlo algorithm used by F^3 is compute bound, with memory requirements that easily fit within all of the platforms' fastest memory resources, I expect that the platforms should exhibit at best parallel scaling close to the ideal parallel scaling factors given in Table 5.3. In the case of the multicore CPUs, this is the number of cores available, for the GPUs, the number of OpenCL compute units, and the FPGAs the parallel instances specified.

Table 5.3.: Ideal parallel scaling of experimental platforms. For CPUs this is the number of cores available, for GPUs the number of OpenCL compute units, and finally for FPGAs the number of parallel instances specified.

Type	Designation	Ideal Parallel Scaling Factor
CPUs	Desktop	8
	Server	16
	Manycore	64
GPUs	NVIDIA Workstation	4
	NVIDIA Cloud	8
	AMD Workstation	12
	Phi	224
FPGAs	ZC706	3
	P385-A7	12
	P385-D5	12
	Max3	9
	Max4	13

Results and Discussion

Figures 5.3, 5.4 and 5.5 present the results of the experiment to characterise the parallel scaling of the heterogeneous implementations generated from F^3. The quantitative values for the experiments may be found in Section A.4.1.

Multicore CPUs: All of the CPUs show strong parallel scaling, as shown in Figure 5.3, with the peak acceleration being greater than the number of floating point computational resources available within each platform (4 in the case of the Desktop platform, 8 in the case of the Server and 32 for the Manycore). This suggests that the implementations are taking advantage of not

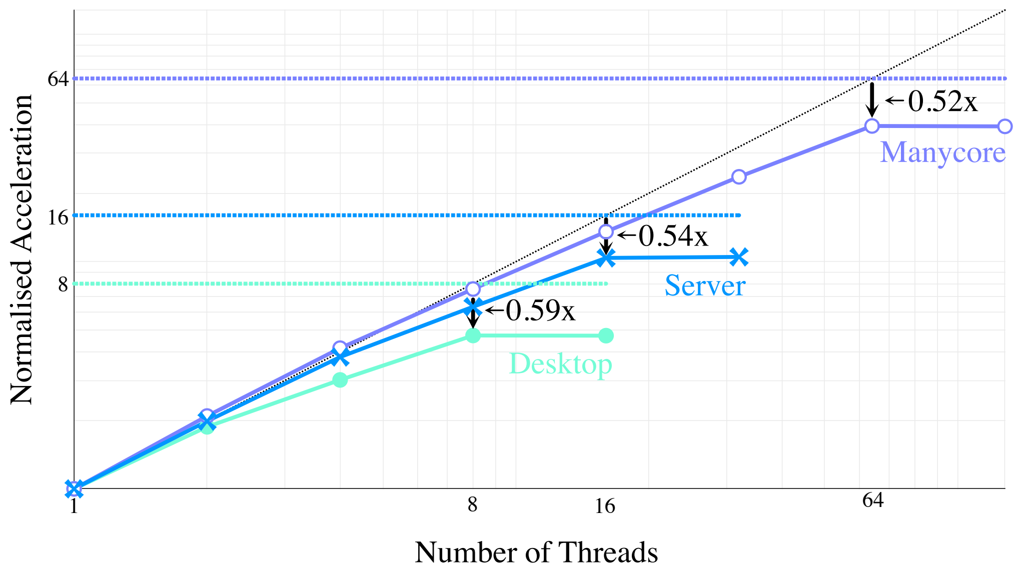


Figure 5.3.: Latency scaling of CPU Platforms as a function of the number of threads executed. The labelled values indicate the factor difference between the ideal parallel scaling and the value achieved for that platform's expected parallelism factor.

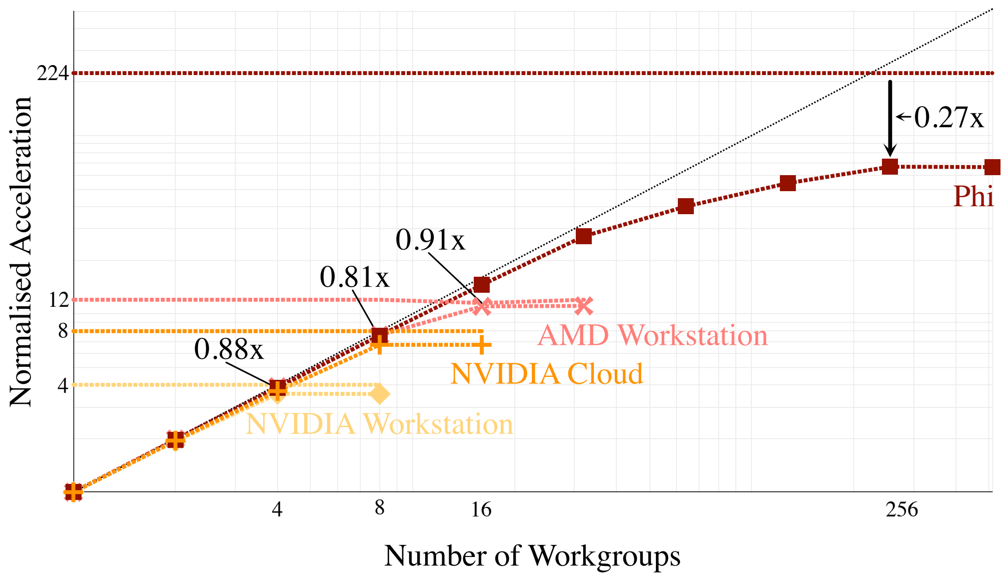


Figure 5.4.: Latency scaling of GPU Platforms as a function of the number of OpenCL workgroups. The labelled values indicate the factor difference between the ideal parallel scaling and the best achieved for that platform.

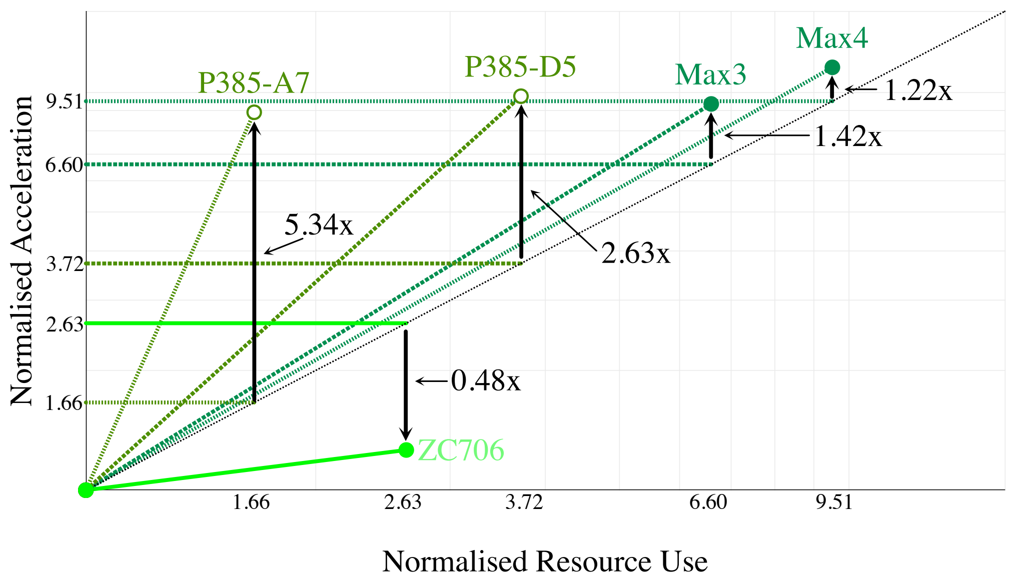


Figure 5.5.: Latency scaling of FPGA Platforms as a function of the normalised resource use. The labelled values indicate the factor difference between the ideal parallel scaling and the best achieved for that platform.

only the parallel arithmetic resources, but also the additional instruction pipelines so as to make the best use of the constrained floating point compute resources available.

GPUs: The Workstation and Cloud GPU platforms in Figure 5.4 also show good linear parallel scaling, although only up to the number of parallel compute resources available, i.e. the number of compute units.

The Phi platform ceases to scale linearly at 64 cores, well short of the 224 compute units reported by the OpenCL runtime. This is explained by the device not actually having 224 physically parallel compute resources as the runtime reports, but in fact 57 [82]. However, by ensuring that the programmer oversubscribes these 57 cores by presenting virtual resources to the system, improved performance is seen, peaking at a 65 times improvement over the single workgroup case on average. This illustrates the hybrid nature of the Xeon Phi well, as this scaling characteristic is similar to that seen in the multicore CPU platforms, where multiple instruction pipelines are used to make full use of arithmetic computational resources. As a result of earlier experiments similar to this, I increased the default factor by which F^3 over-specifies workgroups for Xeon Phis.

FPGAs: All of the FPGA implementations except the ZC706 in Figure 5.5 actually scale better than the linear latency-resource scaling. This is explained by the fact that these implementations have significant resource overhead in the communication and control infrastructure supporting a single accelerator on the FPGA fabric. This resource overhead is then amortised across the parallel compute instances in the larger designs with more parallel compute capability.

However, the fully parallel FPGA implementations took up to 48 hours per option to pass through the FPGA synthesis toolflow, so such super linear scaling comes at significant upfront cost. Hence, in these experiments I only considered two data points per platform - a single instance of the option pricing computation and an implementation that used as much of the device's resources as possible.

The ZC706 platform scales poorly, taking more than twice the number of resources for a less than 20% performance improvement. This is explained by the misalignment of this application, and the intended purpose of the Vivado HLS tools. Vivado HLS is intended for use by embedded computing experts, improving their productivity in designing hardware implementations of software algorithms, and not for accelerating large, compute bound applications. An illustration of this was the difficulty I encountered in implementing the communication between the Vivado HLS FPGA implementation and host CPU, with the tools scheduling communication in such a way that failed to take advantage of parallel optimisations introduced. The resource use scaling though is close to the parallelism factor specified, suggesting that the Vivado HLS implementations have relatively low overhead.

The gradient of the different platforms' improvement provides insight into the efficiency of the different approaches to optimisation. The steeper slope of the P385-A7 and P385-D5 implementations compared to the Max3 and Max4 suggest that the pipeline parallelism-enabling loop unrolling optimisation used by the former makes more efficient use of the FPGA resources than the task parallel instance replication use by the latter. Although, it should be noted that the P385-A7 and P385-D5 are both implemented using the vendor, Altera's, supplied tool, while the Max3 and Max4 are implemented using third party tools from Maxeler, targeting both Xilinx and Altera FPGAs respectively.

From these results, it is clear that the framework is making good use of the parallel computing resources available, scaling strongly when more parallel resources are made available. In the following experiment, I compare these implementations to state-of-the-art implementations from platform programmers.

5.2.3. Inter-Platform Performance Experiment

The second experiment I conducted compared the performance of F^3's option pricing implementations between multiple experimental platform, as well as to two implementations of the same tasks from other researchers.

First, I describe the experiment undertaken. I then make a prediction of the performance of the experimental platforms with respect to a sequential CPU implementation. I conclude the experiment by providing and discussing the results.

Description

Pricing Tasks: Similar to the intra-platform experiment in the previous section, I used the Kaiserslautern option pricing benchmark as well the Black-Scholes Asian option parameters used in the Imperial College work, as given in Table 5.2, as the workload in the experiment. Again, the F^3 implementations performed 10 million simulations per task, with 4096 time steps per simulation path.

External Comparisons: The first set of external implementations that I compared F^3 to was the work of the authors of the Kaiserslautern Option Pricing Benchmark [81]. I also compared

F^3's implementations to that from my colleagues from Imperial College London, who have reported on an implementation of an Arithmetic Asian Option with a Black-Scholes Model-based underlying [80]. In this study, the authors used 1 million simulation paths and only 365 path points, hence, to normalise between the implementations, I scaled the Imperial College results by a factor of 112.22 ($10 \times \frac{4096}{365}$).

Latency Acceleration Measurement: Similar to the intra-platform performance measurement, in the results provided below latency acceleration figures are quoted. However, unlike the previous experiment, the acceleration results for the CPU implementations are the acceleration of the F^3 implementations over F^3's POSIX implementation on the Desktop CPU platform, restricted to a single processor core. The GPU and FPGA platform acceleration results are the performance of those platforms' F^3 implementations over an OpenCL implementation on the Desktop CPU platform, restricted to a single processor core. The Intel OpenCL SDK was used for this OpenCL implementation.

Using a sequential POSIX implementation for the CPU platforms' comparisons, and an OpenCL implementation for the GPUs and FPGAs removes the effect of the differing programming standards upon the results. Hence the acceleration measured, relative to the ideal projections below should reflect the efficiency of the framework and the external implementations.

Throughout this experiment, the single core reference implementations are referred to as *Sequential POSIX* or *Sequential OpenCL* depending upon the programming standard used.

Projection:

I expected the performance of the multicore CPU and GPUs to be in proportion to the theoretical peak performance figures, as demonstrated in Lee et al's work [83], as compared to the theoretical peak performance of a single core of the Desktop CPU platform. I have reported these expected relative acceleration figures in Table 5.4. I did not expect this to be the case for the FPGA implementation, as this calculation does not capture the fine-grained pipeline parallelism inherent in FPGA implementations.

Results and Discussion

Figures 5.6, 5.7 and 5.8 provide the results for the experiments comparing the implementations' performance, including the programmer-created, reference implementations from the researchers at Kaiserslautern [81] and Imperial College London [80]. Latency performance is compared with respect to acceleration over *Sequential*, F^3's POSIX CPU or OpenCL CPU implementations, restricted to only one thread upon the Desktop CPU platform. The full raw throughput performance figures for the experiments may be found in Section A.4.2.

Multicore CPUs: Figure 5.6 illustrates the CPU implementations' performances, including the external, programmer created implementations. The results are consistent with the relative platform characteristics, as given in Table 5.4.

Table 5.4.: Ideal acceleration of experimental platforms relative to a single core of the desktop CPU, as given in Lee et al's work[83].

Type	Designation	Ideal Acceleration over Sequential
CPUs	Desktop	4.00
	Server	8.00
	Manycore	24.00
	KS CPU [81]	4.37
	IC CPU [80]	3.57
GPUs	NVIDIA Workstation	111.09
	NVIDIA Cloud	218.61
	AMD Workstation	113.14
	Phi	178.82
	KS GPU [81]	92.00
	IC GPU [80]	55.54

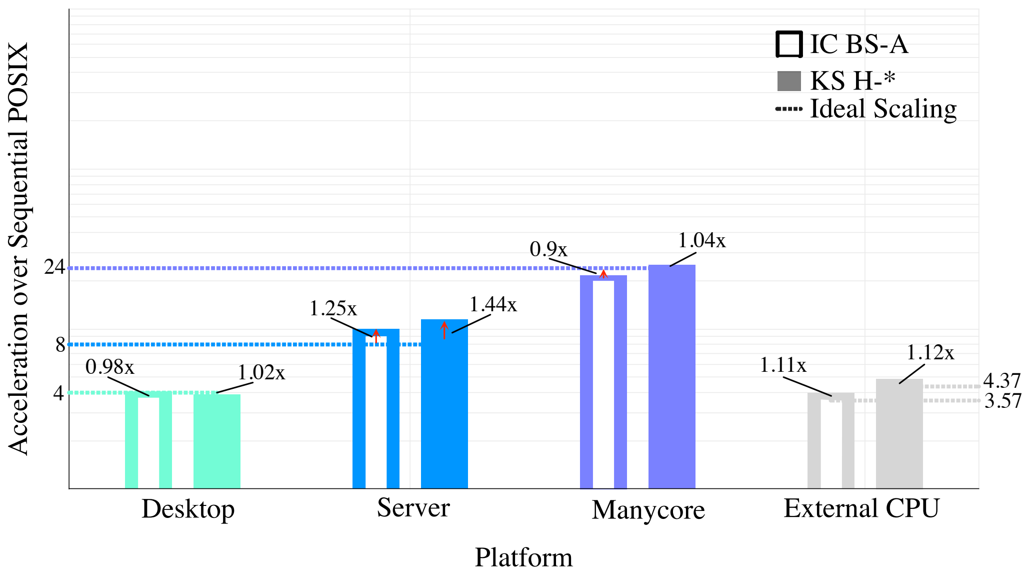


Figure 5.6.: Acceleration of multicore CPU implementations over a single core, implementation on the Desktop CPU platform for Kaiserslautern option pricing benchmark and Imperial College Asian option work. The multiplicative factors indicate the factor difference between the performance measured and the ideal performance as predicted by the methodology employed by Lee et al[83].

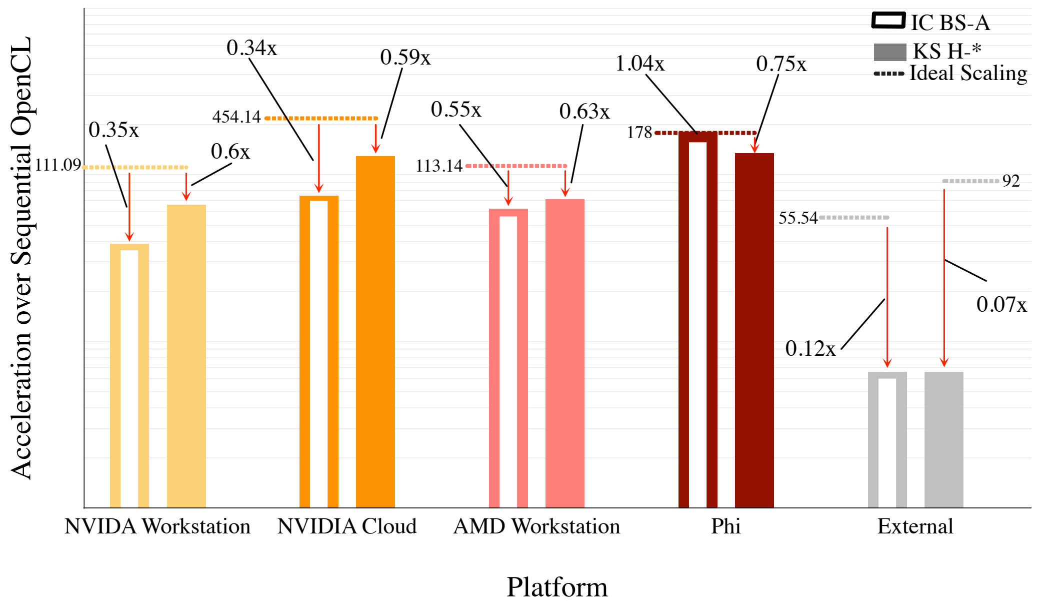


Figure 5.7.: Acceleration of GPU implementations over a single core, OpenCL implementation on the Desktop CPU platform for Kaiserslautern option pricing benchmark and Imperial College Asian option work. The multiplicative factors indicate the factor difference between the performance measured and the ideal acceleration as predicted by the methodology employed by Lee et al.

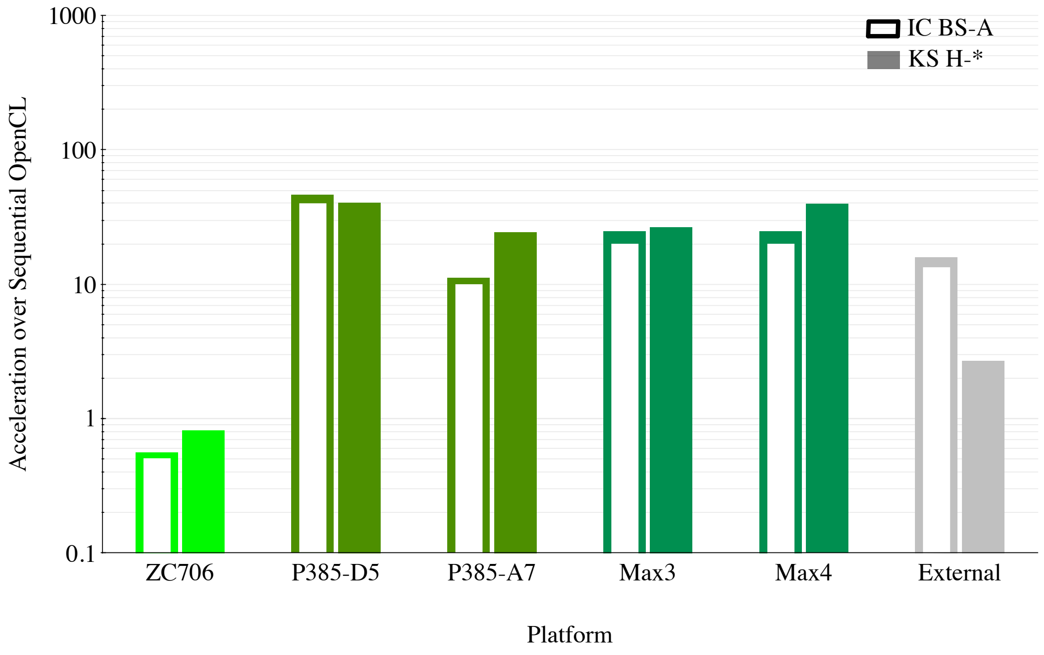


Figure 5.8.: Acceleration of FPGA implementations over over a single core, OpenCL implementation on the Desktop CPU platform for Kaiserslautern option pricing benchmark and Imperial College Asian option work.

An exception is the performance of the Server platform, which did better than the hardware characteristic-based prediction. The Server platform is hosted by Amazon Web Services, and so is a virtualised CPU implemented on potentially many other servers. As a result, it is possible that rather than sharing floating point units, as every 2 cores on the other platforms do, on the Server platform, many threads have sole use of floating point units, and hence outperform the prediction.

The results show that F^3's Desktop CPU implementation are within the same order of relative performance as the external implementations from both Kaiserslautern and Imperial College, although the external, programmer created implementations perform around better, accelerating 10% more than predicted by the hardware characteristics. This result suggests that F^3's generated code is almost as efficient as that written by a programmer, however the code custom written for a specific platform has some advantage.

However, the purpose of the generated code is not to outperform that written by parallel programming experts, but rather to provide access to those resources where there would otherwise not be.

GPUs: Figure 5.7 illustrates the GPU platform results. Generally, the GPU implementations' underperform the predictions based upon hardware characteristics by a factor of 2. This is explained by the difficulty in saturating the high throughput GPU architecture, even though the Monte Carlo algorithm and the OpenCL programming paradigm are well suited to the architecture. This is also despite the copious task and data parallelism that allows for the latency of memory operations to be hidden, as described in Section 5.1.3.

An exception to the under performance trend is the Xeon Phi implementation, which is approaching its projected acceleration figures. This suggests that the code generated by F^3 is well-suited to the hybrid architecture of the Phi, although the efficacy of the Intel OpenCL compiler and native mathematical functions are also potentially a factor.

To explain the difference in performance between the GPU and CPU implementations, I consider the multiplicative difference between the achieved and theoretical peak performance, as described in Lee et al's work[83]. The CPU platforms are achieving about 10% of the theoretical peak performance of the platforms, while the GPUs are achieving almost 25% of the devices' theoretical capacities. However, the Sequential OpenCL implementation is achieving approximately 40% of the CPU's single threaded capacity, hence the GPU platforms appear less efficient by comparison.

Furthermore, this analysis based upon the theoretical performance suggests that the external implementations are dramatically underusing the GPU. A possible explanation is that the work from the other researchers makes use some of the first compute-focused, NVIDIA Tesla GPUs, and hence there have been considerable compiler and architectural innovations in the two generations separating it from the platforms used by F^3. A further consideration is that these researchers were FPGA-focused, and hence maybe did not devote as much time to optimising their GPU implementation.

Of further note is the variability in the Phi's results, with a factor of 3 difference between the best and worst performing tasks[5]. This emphasises the finding of the previous experiment, that the platform's hybrid architecture allows it to execute in an embarrassingly parallel fashion, similar to a GPU, but still use its extra control logic to opportunistically extract compute performance, similar to a CPU.

FPGAs: Figure 5.8 illustrates the FPGA platform results. Unlike the other platforms, the performance of the FPGAs cannot be explained by the task and data parallelism and clock rate. The clocks used in the FPGA implementations are an order of magnitude *slower* than the sequential CPU reference, and the apparent degree of task or data parallelism used is no more than 13. The explanation for the order of magnitude improvement relative to clock rate and apparent parallelism is the fine-grained parallelism enabled by the architecture, algorithm and F^3.

A further influential factor is the efficiency of the parallelism approach used, as described in Section 5.2.2. On this note, relative to device size, the devices using the Altera OpenCL tools do considerably better than those using the Maxeler tools, but the sheer number of resources available to the Max4 platform make it perform comparably to the P385-D5. The poor performance of the ZC706 platform is underlined, with it dramatically underperforming relative to the resources available it, compared to how other implementations use similar resources.

In comparing to the external bodies of work, the differences in the size of the FPGAs used are considerable. The FPGAs I used in my study are of at least one generation more recent, as is the case with the Max3's Virtex 6, or two generations later, in the case of the Stratix Vs used by all of the other experimental platforms [81, 80]. Despite this, the external, programmer implementations are again within the same order of magnitude, suggesting the programmer implementations are achieving considerably more efficient designs. This is not unexpected, given the relative immaturity of the High Level Synthesis tools that F^3 relies upon.

5.3. Heterogeneous Computing Portability

Across the F^3 platforms, I have found that F^3 provides implementations that not only make good use of the parallel computing resources of a platform, but that are also comparable to programmer created implementations.

According to the criteria I laid out in Section 3.1.4, F^3, as an instance of the domain specific approach, is portable, supporting implementation from a single task description to multiple platforms, including multicore CPUs, GPUs and FPGAs. These implementations are also efficient, making good use of the heterogenous, parallel compute resources available, as well as providing comparable performance to implementations created by platform experts.

While not necessarily outperforming platform programmer implementations, F^3 does enable domain programmers, who potentially know very little about the architectures being targeted, to use the parallel compute capabilities of these platforms automatically.

However, I do note that proving this efficient portability criteria is not a novel contribution, rather I have confirmed the results of the bodies of work in Section 2.2.3.

[5]See Table A.11 for a per-task performance breakdown

In conclusion, heterogeneous computing standards, as described in Section 2.1.2, have played an important role in enabling F^3, particularly OpenCL. I understand the success of F^3's implementations as not only evidence of efficiently portable domain specificity, but also as an endorsement of these open standards.

A further observation, which leads into the subject of the next chapter, is the variability of the performance seen, both for a particular platform with different tasks, and between platforms for the same task. There is an interaction between the programming tools used, the computational task specified and the computing platform being used, that is not always predictable ahead of execution. However, this inconsistency in the platform-task performance does open the door to super-linear performance scaling.

6. Predicting Derivatives Pricing

Many financial engineers[1] struggle to make sense of how their tasks will run on the computational platforms available to them. This is because domain programmers are often overwhelmed by the myriad design choices that affect implementation, and often don't know which implementation variable options are relevant, never mind finding the values for those variables that best meet their requirements.

In searching for a way to present these design choices to the domain programmer, I considered the approach of one the most widely used, compute intensive applications: video streaming and computer games. These applications often provide simple sliders or checkboxes which allow users to trade between a variety of domain measures of performance, such as image resolution or responsiveness. Such interfaces are deeply intuitive because the quantified characteristics that are being traded between are easily learnt or understood by the user. The insight behind these interfaces is that the quantified characteristics of the output can serve as abstractions for the implementation design choices. As described in Section 3.2, I have defined these quantified characteristics as domain metrics.

To provide interfaces that allow the domain programmer to balance output metrics according to their requirements, two features are required:

1. Predictive models that map domain function inputs to output domain metrics. These domain metric models need to identify what inputs, which I have defined as implementation variables in Section 3.2.1, can be safely varied as these variables don't affect the correctness of the function's output. In contrast to the domain data type parameters that do.

2. Different metric models need to be reconciled to enable trade-offs between metrics, so that a change in input is reflected in the outputs of the other models. Ideally, any change in value of one metric should be reflected to the domain user in terms of the change to other domain metrics.

In this chapter, I outline how my domain specific methodology can provide a means to model derivatives pricing domain metrics, so as to provide domain programmers with metric trade-offs as a configuration abstraction. To illustrate this claim, I describe my identification and implementation of derivatives pricing domain metrics of latency and accuracy. I then evaluate the validity of these models using F^3, demonstrating that the models can both incorporate further data so as to become more accurate, and also extrapolate, producing useful metric predictions.

[1] and many other types of domain programmers

6.1. Predicting Derivatives Pricing Metrics

In this section I illustrate the predictive domain metric model concept by modelling the important metrics of latency and accuracy for the pricing domain function in derivatives pricing. I first describe the implementation variable and domain metrics in this domain, I then describe how I created models for these metrics. Finally, I describe how I have implemented the latency and accuracy metric metric models using the process outlined in Section 3.2.3, using F^3.

6.1.1. Derivative Pricing Implementation Variables and Metrics

As outlined in Section 3.2.1, I now identify an implementation variable and two domain metrics of the derivatives pricing domain. These metrics provide a means to relate the abstract notions of derivatives pricing, as defined by the domain data types and function given in Section 5.1.1, to the actual implementation of the task upon a real platform.

Implementation Variable

Paths In the Monte Carlo algorithm that I used to implement the derivatives pricing domain function, as previously described in Sections 4.1.3 and 4.2.2, and given in (6.1), N, the number of simulation paths used is an implementation variable. It is an implementation variable by virtue of not being a member of a domain data type, as the other inputs to the algorithm are: r, t are attributes of the underlying data type, whereas T is an attribute of the derivative.

$$V_t = e^{-r(T-t)} \frac{1}{N} \sum_{i=0}^{N-1} V(x_i) \tag{6.1}$$

Outside of the context of the Monte Carlo algorithm, the number of simulation paths has little meaning to the domain programmer, besides maybe appealing to some notion of coverage of the derivative's probability space, as defined by derivative and underlying data types.

Hence a domain specific system, such as F^3, could "safely" specify the value of this variable, provided the programmer was provided with some means of specifying their requirements that the variable impacts upon.

However, within the algorithm it directly impacts the statistical properties of the computation's result, such as the size of the result's confidence intervals [74]. From an implementation point of view, the number of paths determines the scale of the computation, and also gives an upper bound on the practical task and data parallelism of that particular option pricing task.

Domain Metrics

Latency The latency between when a pricing operation is initiated and when it returns a price is fundamentally important within the financial domain [74]. There is a practical limitation that if a calculation takes too long, the assumptions that the financial engineer used to set the underlying and derivative values might be superseded by actual events, such as a change in value of an exchange rate. Furthermore, the time at which prices are received affects how traders use those prices. Minimising the latency of the pricing operation is desirable, as this confers first-mover advantage.

Accuracy In the financial domain, the accuracy of a computed price is often expressed in probabilistic terms. When using the Monte Carlo algorithm, often the 95% confidence interval is used, which gives the size of the finite interval around the computed price for which there is a 95% confidence that the true convergence value of the algorithm[2] lies within that interval. As small a confidence interval as possible is desired, as this means less risk has to be accounted for, and hence the derivative may be hedged on a smaller margin.

6.1.2. Latency and Accuracy Metric Models

As described in Chapter 4, derivatives pricing is the only domain function in the derivatives pricing domain. In this subsection, I develop the metric models, as per Section 3.2.2, for the domain metrics of latency, (6.2), and price accuracy, (6.3), for the pricing function in terms of the number of simulations paths implementation variable as implemented using the Monte Carlo algorithm.

Latency Model

I have used a simple, linear latency metric model in (6.2), a function of a single implementation variable, the number of paths (n), i.e as per (3.2), $V = n$, $\mathcal{V} = \mathbb{Z}_+$, and (6.1).

The linear nature of the model reflects the $O(N)$ complexity of the Monte Carlo Algorithm. The model's coefficient (β) translates to the time spent per Monte Carlo path. Similarly, γ, the constant component of the latency metric model, captures the fixed time spent initialising the computation, as well as any time spent communicating the task to and returning the result from the target platform.

$$f_L(n) = \beta n + \gamma. \tag{6.2}$$

Accuracy Model

The accuracy metric model that I have used is based upon the convergence of the Monte Carlo algorithm, which is given by the inverse square root of the number of paths, scaled by a coefficient (α). The model is given in (6.3).

$$f_C(n) = \frac{\alpha}{\sqrt{n}}. \tag{6.3}$$

Unified Model

To relate the two domain metrics of latency and accuracy, I solve for n and use it to express (6.2) and (6.3) as a trade-off between the latency and accuracy, as given in (6.4). This means that n, V in this implementation can be found for a given accuracy ($c \in \mathbb{R}_+$).

$$f_L(c) = \frac{\delta}{c^2} + \gamma. \tag{6.4}$$

where:

[2]In accordance with the model parameters specified. Whether the true value of the model reflects reality is the concern of the financial engineer and anyone affected by their behaviour.

Listing 6.1: Implementation of metric model generation within F^3, assuming the option definition given in Listing 5.3

```
#Creating the platforms and solver objects
multicore_cpu = Multicore_CPU()
mc_solver_cpu = MonteCarlo([KSO_1,KSO_2], multicore_cpu)

#Calling the CPU solver to generateand compile
mc_solver_cpu.generate()
mc_solver_cpu.compile()

#Populating the model
mc_solver_cpu.populate_models()

#predicting the latency and accuracy for 10 Million Paths
pred_latency = mc_solver_cpu.latency_model(1e7)
pred_accuracy = mc_solver_cpu.accuracy_model(1e7)
```

$$\delta = \beta\alpha^2.$$

6.1.3. Implementation in F^3

In order to bring the ability to model domain metrics into F^3, I exploited the automated compilation and execution capabilities of the framework, as described in Section 5.1.2. I used these capabilities to build the online benchmarking process, as described in Section 3.2.3, into the base Monte Carlo solver class that the platform specific solvers inherent from, thus implementing the models only once, but reusing them for each type of computational platforms. If I were to find a platform specific model that was more accurate, I could merely override the base Monte Carlo model provided with this more accurate model for that platform's implementation[3].

If a platform specific algorithmic object has been created, the corresponding method may be called to perform the benchmarking process for the option pricing problems currently allocated to it, upon the platform to which it is targeted. Upon completion of the benchmarking process, a linear algebra library, SciPy [84], is used to perform the least squares regression and the corresponding latency and accuracy model is associated with that particular solver instance.

An example of a solver using the prediction capability built into the framework can be found in Listing 6.1.

6.2. Evaluation

Having demonstrated the efficient portability of the framework in Section 5.2, similarly in this section I evaluate the framework's ability to model latency and accuracy domain metrics upon for a broad set of problems upon a large set of experimental platforms.

[3]Thus far however, I have not.

To evaluate the claim that the domain metric models are able to characterise tasks on heterogeneous platform, I need to prove the following two properties, as outlined generically in Section 3.2.4, for a diverse set of platforms and tasks for both of the models I developed in Section 6.1.2:

- *Incorporation*: As the set of benchmarking data points grows, the domain metric value predicted by the model converges on the value of the domain metric measured at run-time for the same inputs.
- *Extrapolation*: For a given amount of benchmarking, the domain metric values predicted by the models remains "reasonably" close to those measured at run-time for a large set of simulation path values. What is reasonable in this context I have heuristically defined as a less than 10% change for an order of magnitude paths than benchmarking.

To evaluate both criteria, I have measured the relative error as given in (3.8), adapted to this context in (6.5), where the absolute difference between the measured run-time value ($f_k(n)$) and the predicted metric value ($\hat{f}_{k,b}(n)$) after b benchmarking data points for n simulation paths , is divided by the run-time value. The run-time metric value is measured when the task is run with the specified option pricing task inputs and implementation variable, the number of simulation paths (n). I have measured this for both the latency and accuracy models described above across diverse sets of heterogeneous computing platforms and tasks.

$$r_k = \frac{\left| f_k(n) - \hat{f}_{k,b}(n) \right|}{f_k(n)} \tag{6.5}$$

6.2.1. Experimental Setup

In this subsection I describe the broad sets of heterogeneous platforms and tasks that I used to test the incorporation and extrapolation properties of the latency and accuracy metric models for my derivatives pricing domain case study. For both sets, I describe the nature of the heterogeneity. The intention is that in a similar manner to the portability feature, that by evaluating a broad set of platforms and tasks, I make a strong case for the efficacy of the domain metric models.

Heterogeneous Platforms

An overview of the heterogeneous platforms that I used is given in Table 6.1.

The first class of platform heterogeneity is device type - I have made use of a wide array of Multicore CPUs, GPU and FPGA-based computational platforms. The second dimension is the manufacturer within each device category, which I have varied as much was practically possible. The final class is the diversity of interconnections used between the computational platforms, achieved with varied geographic locations.

I describe the detailed compute capabilities of the experimental platforms in Tables **??** and A.5. As the Monte Carlo algorithm being used is amenable to parallel execution, I expected that the GPUs and FPGAs would provide the best application performance. This is also reflected by the results reported in Section 5.2.

Table 6.1.: Overview of heterogeneous computing platforms used in domain modeling evaluation experiments. Order within device categories is by network latency.

Device Category	Platform Designation	Platform Name	Network Location Name
CPUs	Desktop	Intel Core i7-2600	Localhost
	Local Server	AMD Opteron 6272	ICL Datacentre
	Local Pi	ARM 11 76JZF-S	ICL Insecure subnet
	AWS Server EC1	Intel Xeon E5-2680	AWS East
	AWS Server EC2	Intel Xeon E5-2670	AWS East
	GCE Server	Intel Xeon	GCE Central
	AWS Server WC1	Intel Xeon E5-2680	AWS West
	AWS Server WC2	Intel Xeon E5-2670	AWS West
	Remote Server	Intel Xeon E5-2680	UCT Datacentre
GPUs	Local GPU 1	AMD FirePro W5000	ICL EEE Workshop
	Local GPU 2	NVIDIA Quardo K4000	ICL EEE Workshop
	AWS EC GPU	NVIDIA Grid GK104	AWS East
	AWS WC GPU	NVIDIA Grid GK104	AWS West
	Remote Phi	Intel Xeon Phi 3120P	UCT Datacentre
FPGAs	Local FPGA 1	Maxeler Max 3424A	ICL EEE Workshop
	Local FPGA 2	Nallatech P385-D5	ICL EEE Workshop

An important caveat however is that the platform performance reflect implementations produced by F^3, in addition to reflecting the inherent capabilities of the devices. I have also provided the network latency for each platform in Table A.7. The computational characteristics of the tasks are also described in Table 6.2.

I expect the compute capabilities to determine the coefficient of the latency model in (6.2), β, while the network latency will largely determine the constant coefficient, γ. Particularly prominent data-points are the Remote Server and Phi, which have orders of magnitude longer communication times than the other platforms. Another notable outlier is the Local Pi platform, which is orders of magnitude less capable than the next most computationally powerful platform.

Option Tasks

Table 6.2 provides a breakdown of the broad set of derivative pricing tasks that were used to evaluate the domain metric models. In addition to the types of underlying and derivatives used, the total amount of computational work for each task is specified in Table A.8.

The domain parameters for the pricing task operations, as defined in Section 3.2.1, such as the proprieties of underlying model, were generated using uniform random numbers within the values from the Kaiserslautern option pricing benchmark. A rejection procedure was utilised to keep the relative magnitude of the pricing tasks within the same order of magnitude.

This workload reflects a diverse array of tasks with the pricing task category, with those using the Heston underlying being more than twice as computationally intensive as the Black Scholes-based pricing tasks. Similarly, the European and Asian option tasks have less complex

Table 6.2.: Overview of domain model evaluation workload of option pricing tasks.

Task Name	BS-A	BS-B	BS-DB	BS-DDB	H-A	H-B	H-DB	H-DDB	H-E	Total
Number	10	10	10	5	25	29	29	5	5	128

control flows than the barrier options. I weighted the workload of 128 tasks towards the more computationally intensive tasks, as these produce more varied domain metric characteristics.

6.2.2. Latency Metric Model Error

The latency model incorporation results are given in Figures 6.1, 6.2, 6.3 and 6.4. Figures 6.5, 6.6, 6.7 and 6.8 give the results of the latency extrapolation experiments. I evaluated the latency models on a platform basis, i.e. the geometric mean of the error of all task on that platform, as per the entries in Table 6.1, as well as the geometric mean of the three platform categories, as given by the first column in the table. I expect the latency models to be specific to a particular task-platform pair, and hence I have considered each task upon each platform.

The error is expressed as a function of the ratio of the number of paths used for benchmarking (b), and the number of paths used at runtime (n). In all cases only the mean error is given, as the deviation from the mean is to small to indicate.

In the interpolation experiments, as the ratio approaches 1, the number of benchmark paths becomes equal to the run-time paths, and the error in the prediction is expected to be minimised, as the prediction model has complete information. This reflects that the amount of benchmarking is being varied, while the number of paths at run-time are fixed.

In the extrapolation experiments, as the number grows above 1, this reflects that the number of run-time paths is many multiples of the number of benchmarking paths used. This reflects that in this experiment, the number of run-time paths is varied, with a fixed number of benchmarking paths.

Latency Model Interpolation

Multicore CPUs In Figure 6.1, the CPU latency model error is presented. As more information, i.e. the benchmark to run-time ratio becomes closer to one, is made available to the CPU platform models, the relative error becomes smaller. Of note is the error of the Local Pi platform, which is an order magnitude less than the next platform. The Local Pi doesn't have any ability to compute in parallel, having only a single computational platform, hence this suggests that the error in the latency model is being introduced in the allocation and scheduling of threads to different parallel compute elements within the systems. This is confirmed by considering the platform with the highest error, the Local Server, which not only has 16 cores per CPU, but also multiple CPUs.

GPUs Figure 6.2 provides the GPU latency model error. The error of both the Local and AWS GPUs illustrate a reliable incorporation trend, similar to the CPU platforms, with the

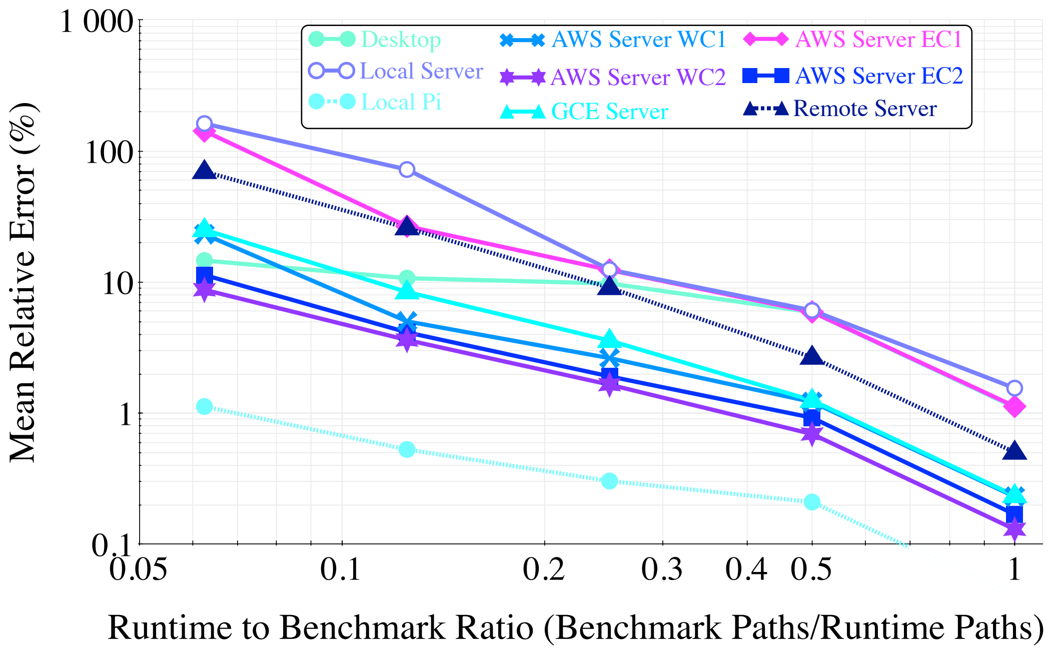


Figure 6.1.: Relative error of multicore CPU latency models for a fixed run-time number of paths that take 4.69 seconds to execute and varying numbers of benchmark paths.

error declining in a linear fashion given further benchmarking information. The Remote Phi platform is less convergent, with the error being a multiple of the next highest error in many cases. This is due to the high latency of the platform's network connection, making up most of the run-time. As a result, any small change in the network latency, which is inherently volatile, had a corresponding impact upon the quality of the latency model created.

FPGAs Figure 6.3 provides the FPGA latency model error. Both platforms also exhibit a linear scaling, with the error of the 1st Local FPGA being considerably higher than those seen for the CPU and GPU platforms.

I initially found this result surprising, as I expected that the FPGA platforms would provide a dedicated architecture with reliable timing characteristics. However, the platforms are susceptible to variations in scheduling of communication by the operating system, and the Maxeler platform is particularly prone to this variation. Another source of latency variability is the configuration of the FPGA device itself. Similar to network communication in the Remote Phi, this makes up a considerable length of time on both platforms. This is of variable length, as if the required bitstream has already been configured on the device, than reconfiguration is not necessary, whereas if it is not, then the time consuming reconfiguration procedure is necessary.

Platform Categories The interpolation results are summarised in Figure 6.4. The platform category results illustrate that as the benchmarking data set grows, the models become more accurate. This suggests that the incorporation property holds for the latency model.

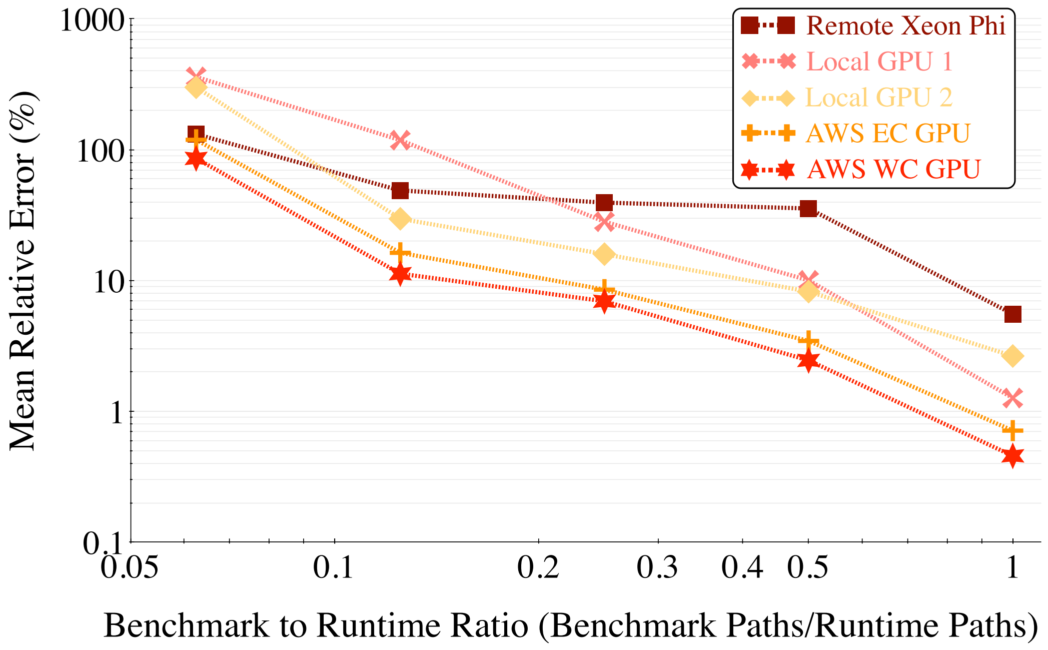


Figure 6.2.: Relative error of GPU latency models for a fixed run-time number of paths that take 4.69 seconds to execute and varying benchmark time

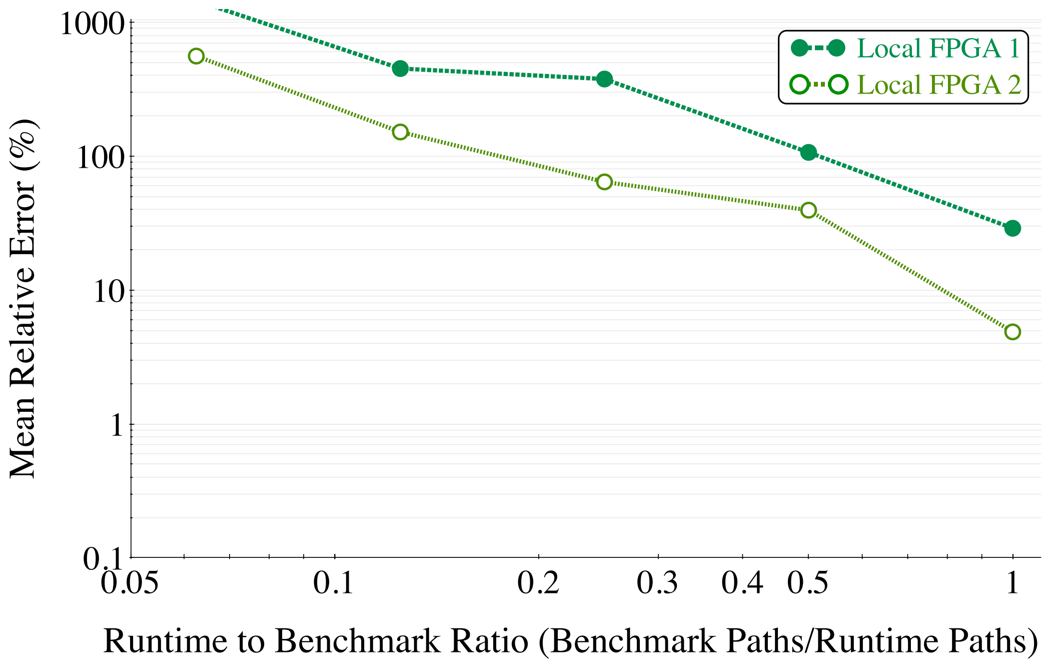


Figure 6.3.: Relative error of FPGA latency models for a fixed number of run-time paths that take 4.69 seconds to execute and varying benchmark time

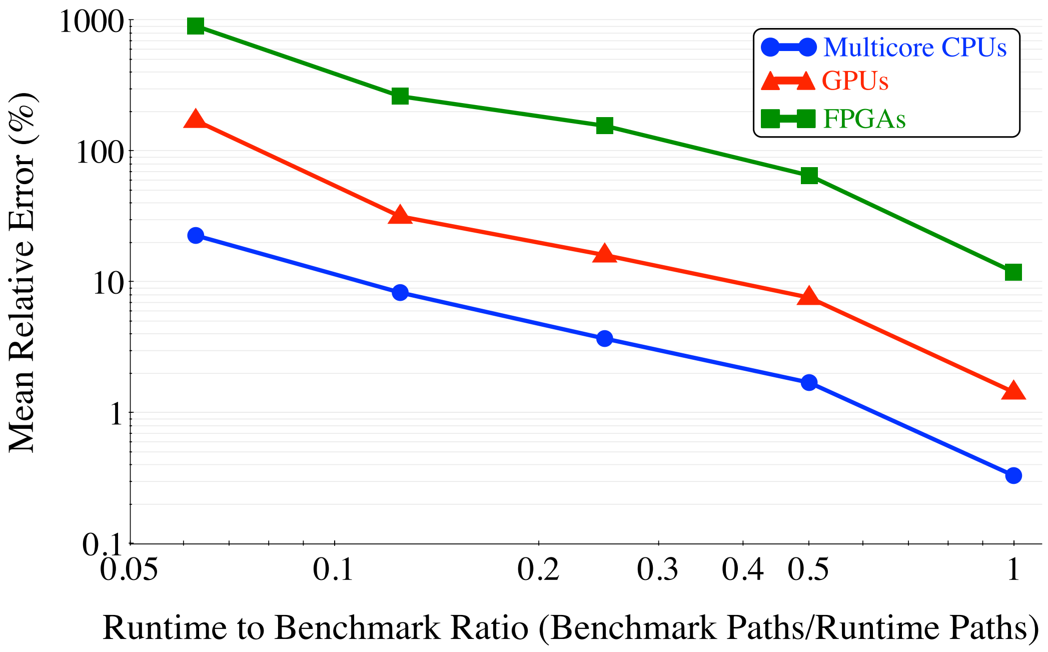


Figure 6.4.: Relative error of platform category latency models for a fixed number of run-time paths that take 4.69 seconds to execute and varying benchmark time

Extrapolation

Multicore CPUs Figure 6.5 illustrates that as the problem size is scaled up, the latency model continues to predict the run-time latency well. After a small initial increase, the error for most platforms stays close to 1%. An exception is the Remote Server, where the error grows linearly. This is due to the high network latency of the server results resulting in a latency model with relative few benchmarking data points initially, and hence an inaccurate model.

GPUs Figure 6.6 echoes the trend seen with the CPU platforms. After a small initial increase, the error of most of the GPU platforms remains stable, close to 1%, even decreasing slightly. This is due to the primary source of error, OS and subsystem scheduling overhead, contributing less to the overall run-time. Although, again, the Remote Phi platform exhibits a high error characteristic due to a low number of benchmarking data points.

FPGAs Figure 6.7 also reflects the trend seen in CPU and GPU platforms. After a small initial increase, the error remains stable, although the FPGA cases this is closer to 10% than 1%. This is due to the higher variability seen by these platforms, hence the models are more inaccurate to begin with.

Platform Categories The extrapolation results are summarised in Figure 6.8. The platform category shows how the models scale as the run-time prediction target is increased for a fixed benchmarking time of 4.69 seconds per task or 10 minutes in total, and an increasing run-time target. These results demonstrate that for a run-time target of more than an order of magnitude greater than the benchmarking procedure, the latency models are capable of extrapolating, making predictions with less than 10% error in most cases, the heuristic criteria I defined. The

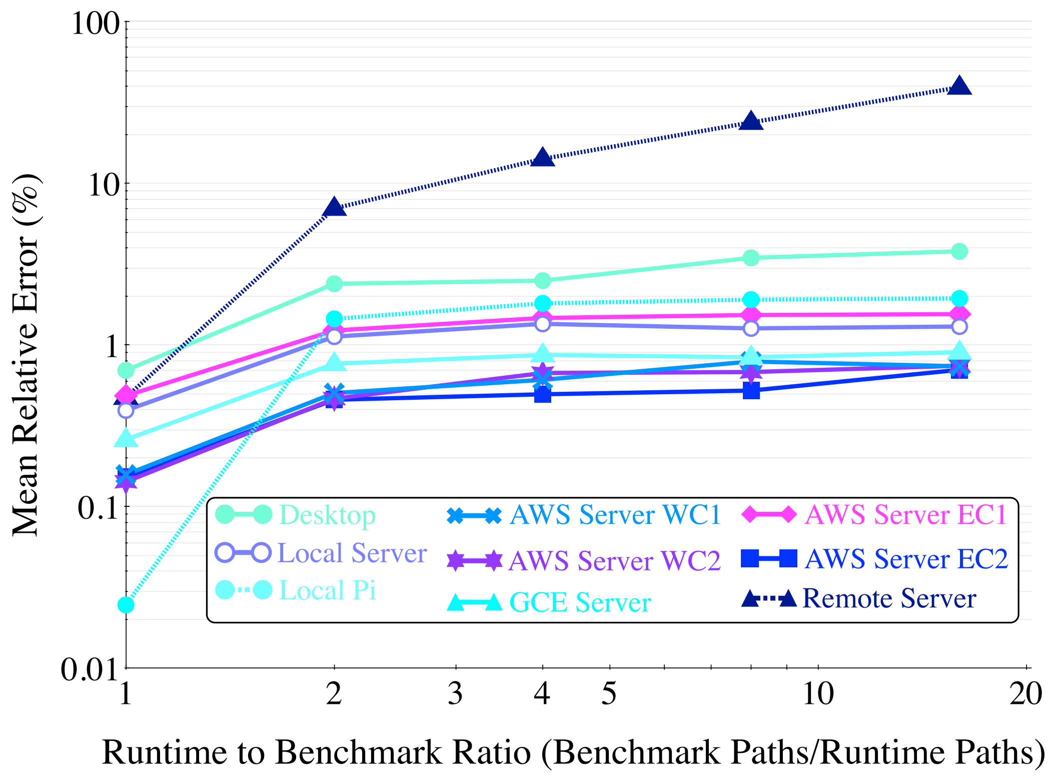


Figure 6.5.: Relative error of CPU latency models for a fixed number of paths that take 4.69 seconds to execute and varying run-time paths.

remote Phi and server models' poor performance is explained by the benchmarking time being too short to accurately solve for the true coefficient and constant values.

6.2.3. Accuracy Metric Model Error

The accuracy model results are given in Figure 6.9 for the incorporation, and Figure 6.10 for the extrapolation. The accuracy model results are presented as minimum, geometric mean and maximum of the model results within the pricing task categories in Table 6.2 as the axes of the plots. This is because the accuracy of the task is not affected by the platform of implementation, but rather the number of simulation paths used and the problem parameters.

Similar to the previous experiment, the error is expressed as a function of the benchmark paths to run-time paths ratio, as described in Section 6.2.2.

Interpolation

Figure 6.9 illustrates that as information is added to the benchmarking procedure, the relative error in the accuracy prediction model decreases across the different tasks categories. This is explained by the convergence of the Monte Carlo algorithm being a proven property, hence a low number of data points are required to solve for the convergence coefficient, α, as given in (6.3). Some of the Heston option tasks present a relatively high maximum error, however, as can be seen by the task category geometric mean these errors average out close to 10%.

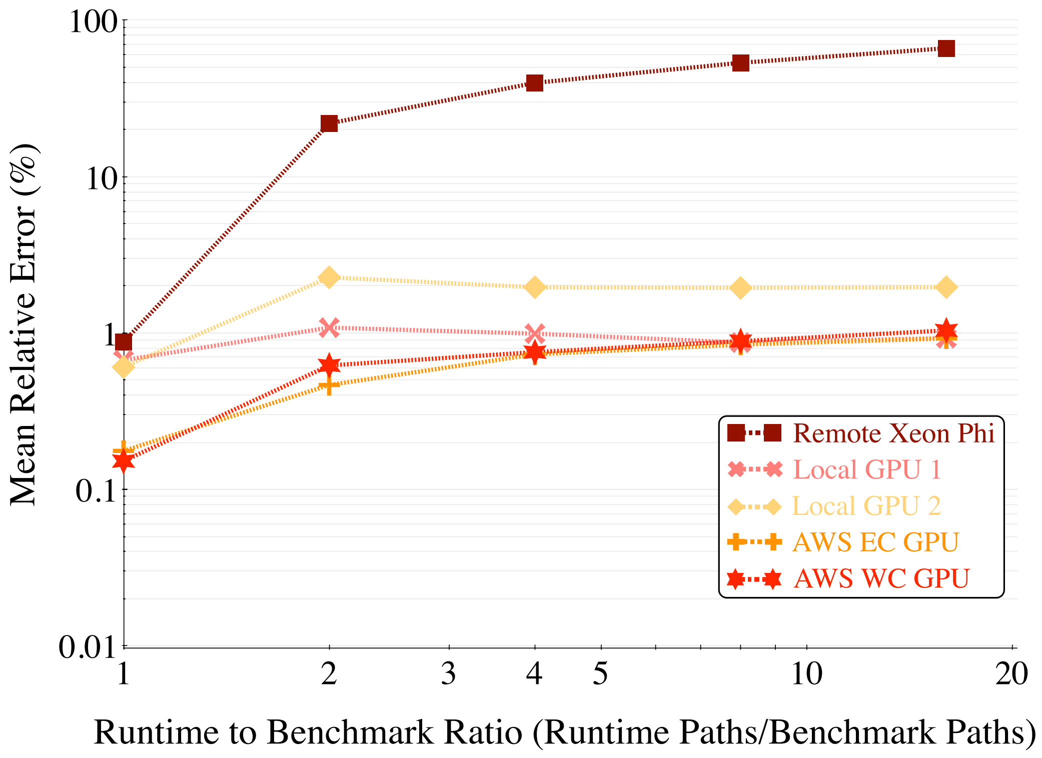


Figure 6.6.: Relative error of GPU latency models for a fixed number of paths that take 4.69 seconds to execute and varying run-time paths.

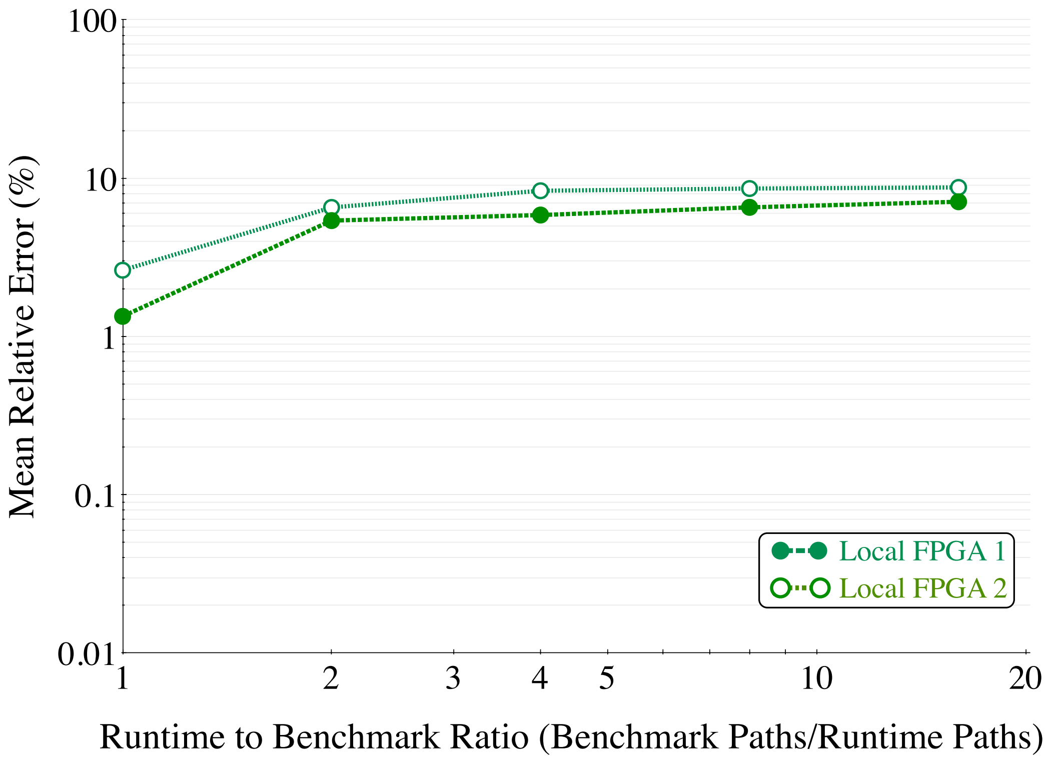


Figure 6.7.: Relative error of FPGAlatency models for a fixed number of paths that take 4.69 seconds to execute and varying run-time paths.

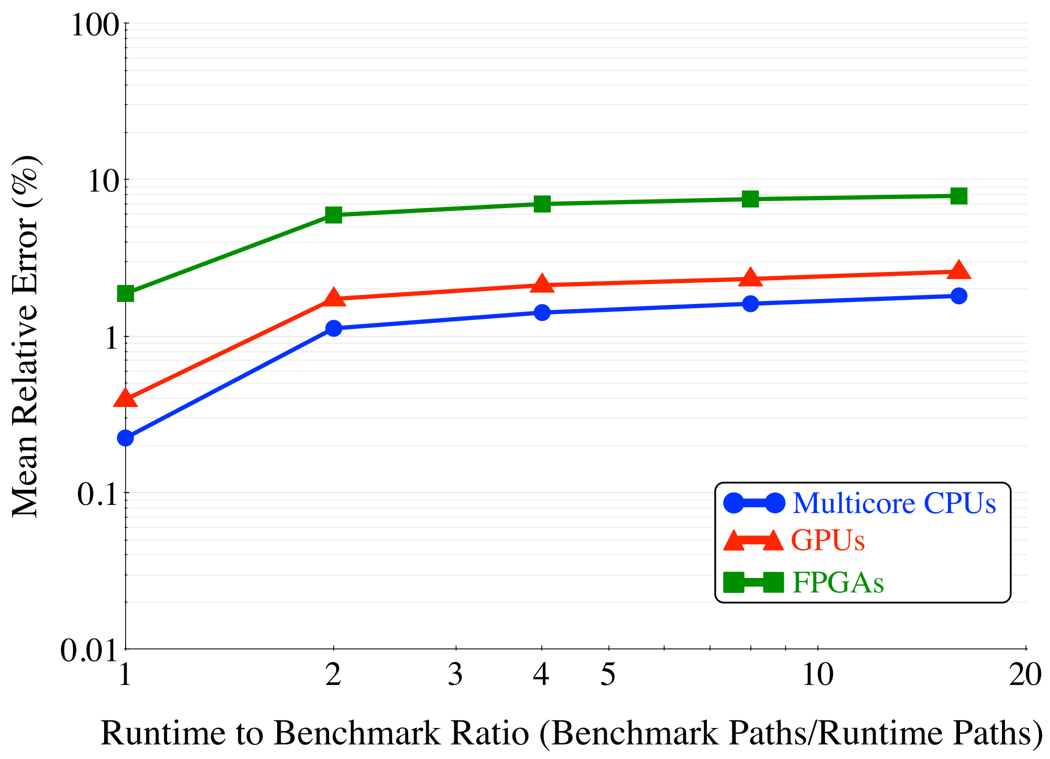


Figure 6.8.: Relative error of platform category latency models for a fixed number of paths that take 4.69 seconds to execute and varying run-time paths.

Extrapolation

Figure 6.10 shows how the models scale as the run-time target is increased. Similar to the Latency model results, the models scale well for more than an order of magnitude, with relatively little change in the error minimum, geometric mean and maximum.

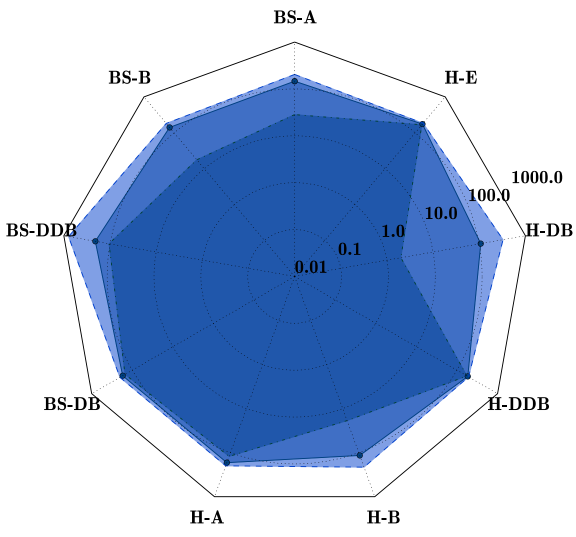
In all cases, the mean error for all task categories remains within the 10% heuristic criteria I defined as being acceptable for extrapolation.

6.3. Modelling Domain Metrics on Heterogeneous Platforms

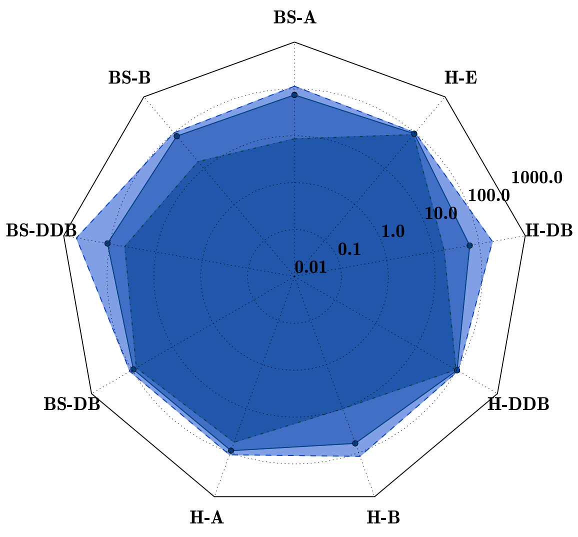
For the domain of derivatives pricing, I have shown that predictive domain metric models for latency and accuracy can be found and implemented.

As Figures 6.4, and 6.9 indicate, the latency and accuracy metric models incorporates new data, hence becoming more predictive and converging on the run-time metric value. As Figures 6.8 and 6.10 reflect, the models also extrapolate well, with a relatively minor increase in latency and accuracy error for run-times of more than an order of magnitude longer than the benchmarking time.

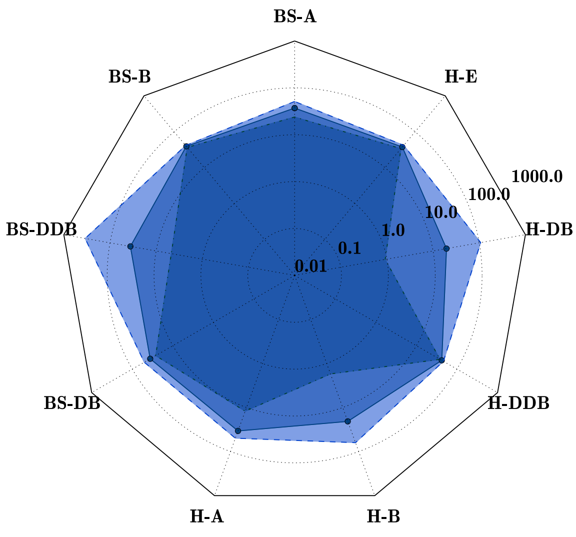
This work builds upon the previous chapter, making use of the efficient, portable execution enabled by the domain specific approach to automatically perform the process of populating these prediction models for the domain function. Hence the existence of this feature is testament to the value of the previous one.



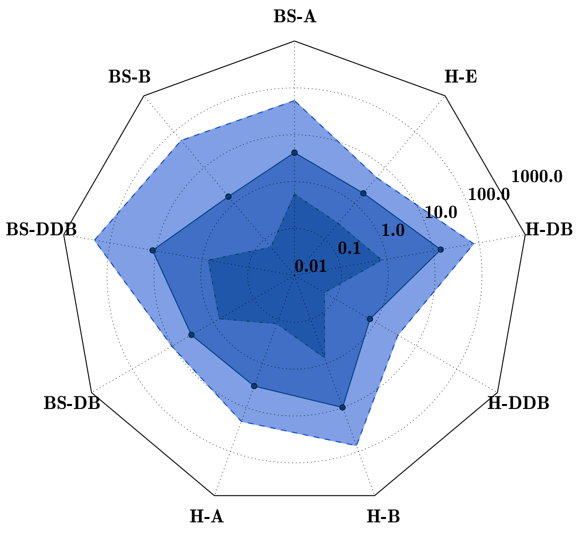
(a) 0.125:1.0



(b) 0.25:1.0

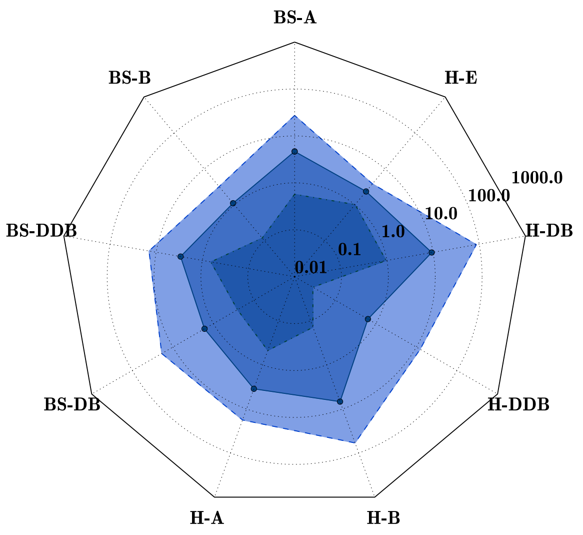


(c) 0.5:1.0

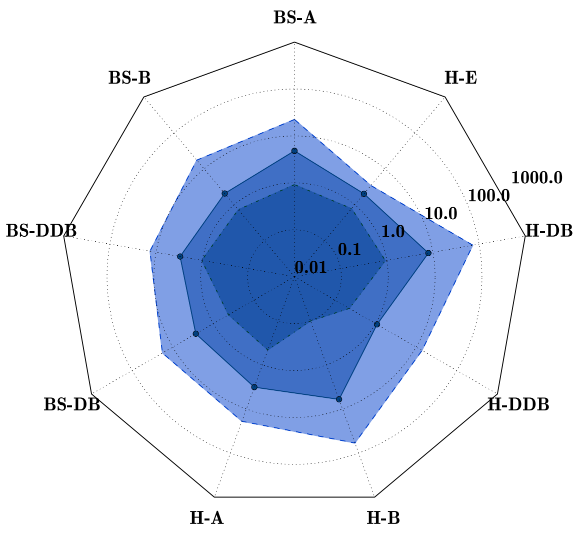


(d) 1.0:1.0

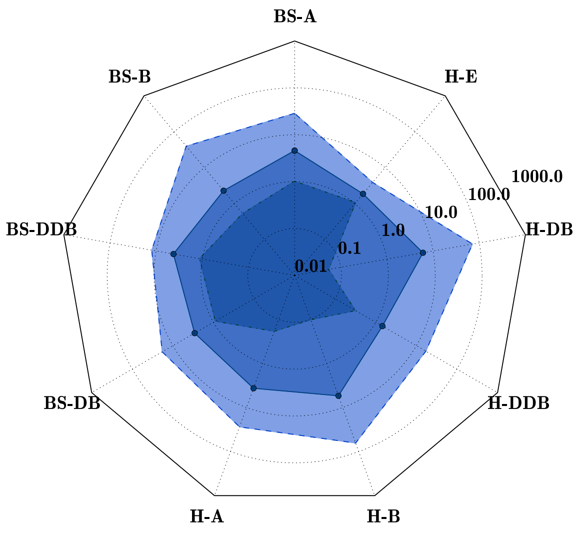
Figure 6.9.: Relative error of accuracy models for a fixed number of run-time paths that take 4.69 seconds to execute and varying numbers of benchmark paths. Ratio is expressed as *Benchmark Paths*:*run-time Paths*. The innermost region represents the minimum error, the middle region the geometric mean relative error and the outermost the maximum.



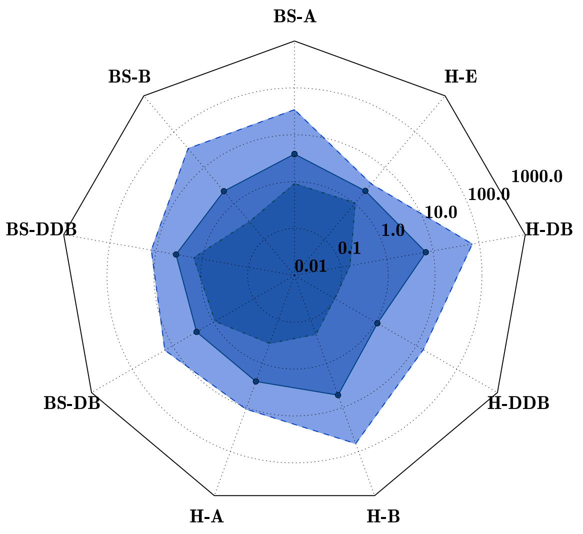
(a) 1:2



(b) 1:4



(c) 1:8



(d) 1:16

Figure 6.10.: Relative error of accuracy models for a fixed number of benchmark paths that take 4.69 seconds to execute and varying numbers of run-time paths. Ratio is expressed as *Benchmark Paths*:*run-time Paths*. The innermost region represents the minimum error, the middle region the geometric mean relative error and the outermost the maximum.

The existence of the models for two metrics, with both the incorporation and extrapolation properties gives weight to the claims made in Section 3.2. As I have described in Section 2.3.1, I have found the characterisation of tasks with respect to computational platforms to be an understudied area that is usually conflated with other areas such as broader application studies or workload partitioning. Hence my proposed approach to the implementation and evaluation of a domain specific means for relating tasks and platforms is a modest contribution in addressing this gap in computing engineering scholarship.

In the next chapter, I refine these models so that multiple tasks and platforms may be considered. I then show how these multiple-platform and task models then allow for my domain specific methodology to automatically partitioning work across the available resources in an optimal fashion.

7. Partitioning Derivatives Pricing

The previous chapter described how derivatives pricing tasks and platforms may be characterised using relatively simple domain metric models for accuracy and latency. Such models would be useful for a financial engineer balancing a decimal place of accuracy against the additional hour of execution it would require, or as an evaluation tool for a bank's CTO faced with a growing array of heterogeneous procurement options. However, a far more common use case is when there are portfolios of multiple derivatives that need to be valued, and multiple heterogeneous computing platforms upon which these derivative pricing tasks could be executed.

In a more general scenario of many tasks and platform, the central challenge is how the workload can be partitioned across the available platforms while improving, or at least maintaining, the metric trade-off characteristic achieved in the previous chapter. Stated in terms of the domain models, how can the domain specific approach be extended to cope with multiple tasks and platforms? In this chapter, I address this question by describing how the performance predictions made by the derivatives pricing domain metric models can be combined using task and platform reduction functions, as described in Section 3.3.1. By being able to combine the metric values, this allows for the partitioning of work across the available resources to be formulated as an optimisation problem that can be solved automatically, i.e. without domain programmer intervention, in an optimal fashion.

I first describe how the unified latency and accuracy models described in Section 6.1.2 can be incorporated into a formulation of the partitioning problem described in Section 3.3. I then detail several different approaches, and the implementation thereof to find solutions to this optimal allocation optimisation. In the second section of the chapter, I provide an evaluation of the partitioning approach, using both synthetic task and platform data, as well as the option pricing problems and heterogeneous platforms used in the previous chapter.

7.1. Derivatives Pricing Task Partitioning

I now formulate the derivatives pricing workload partitioning problem using the latency and accuracy metric models, as well as outline and describe the implementation of three approaches for solving the resulting problem. Finally I describe how I have implemented these three partitioning approaches within F^3.

7.1.1. Reformulating the Partitioning Problem

In (7.1) the unified latency-accuracy domain metric model I described in (6.4) has been applied to the general, constrained allocation problem specified in (3.16). As described in Section 3.3.2, the relaxed form of the problem may be used, as the degree of parallelism is sufficiently large that the allocation variable, $\boldsymbol{A}$ can be expressed as a proportion of the problem being solved on a particular platform.

The vector $\vec{c}$ gives the task accuracies, with each element corresponding to the required 95% confidence interval for each task, while $\boldsymbol{\gamma}$ is the task-platform constant matrix. Similarly, $\boldsymbol{\delta} : \vec{c}^2$ is the element-wise division of the delta coefficients by the required accuracies of the tasks. In this case, an additional constraint to ensure the specified accuracies is not required, as the unified metric model has already captured this constraint implicitly.

$$\begin{aligned} \underset{\boldsymbol{A}\in\mathbb{R}_+^{\mu\times\tau}}{\text{minimise}} \quad & G_L(\vec{F}_L(\boldsymbol{A},\vec{c})) \quad \vec{c}\in\mathbb{R}_+^{\tau} \\ \text{subject to} \quad & \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1,2,\ldots,\tau. \end{aligned} \tag{7.1}$$

Where:

$$\begin{aligned} G_L(\vec{F}_L(\boldsymbol{A},\vec{c})) &= \max(\vec{F}_L(\boldsymbol{A},\vec{c})), \\ \vec{F}_L(\boldsymbol{A},\vec{c}) &= (\boldsymbol{\delta} : \vec{c}^2 \circ \boldsymbol{A} + \boldsymbol{\gamma} \circ \lceil \boldsymbol{A} \rceil) \cdot \mathbf{1} \qquad \boldsymbol{\delta} \in \mathbb{R}_+^{\mu\times\tau}, \boldsymbol{\gamma} \in \mathbb{R}_+^{\mu\times\tau}. \end{aligned}$$

An important feature of the formulation given in (7.1) is its non-linearity as a result of the ceiling function in $\vec{F_L}$. This reflects (6.4), in that there is a constant set up time for each task upon a platform, regardless of the amount of the task allocated.

7.1.2. Partitioning Approaches

I have investigated three approaches to solving this program from the categories identified in Section 2.3.3. In each case, I have also commented upon how likely it is that I would expect a domain programmer to arrive at such an approach in the absence of a domain specific methodology.

Heuristic Allocations

Below I define two heuristic solutions to solving the program specified in (7.1), the best platform and proportional allocation heuristic. Both are intuitive and have a low computational complexity. My opinion is that competent domain programmer to formulate something similar to these heuristics within a day or a week.

The heuristics are by no means the most sophisticated, considering the exhaustive evaluation given in Braun et al's work [40]. However, heuristic represent the two most direct ways to use heterogeneous computing resources - use the single best platform, or allocate work linearly, i.e. in proportion to the platforms' capabilities. My intention is to use these heuristics as a baseline for my investigation in workload partitioning.

Best Platform Heuristic The first best platform heuristic I propose in (7.2) is intuitive: all of the tasks are allocated to the single platform that completes all the tasks with the shortest makespan.

$$\begin{aligned}
&\vec{L}_i < \vec{L}_x && i, x = 1, 2, \ldots, \mu, i \neq x, \\
&A_{i,j} = 1 && j = 1, 2, \ldots, \tau, \\
&A_{x,j} = 0,
\end{aligned} \tag{7.2}$$

Where:

$$\vec{L} = \vec{F_L}(\mathbf{1}, \vec{c}).$$

Proportional Allocation Heuristic The second heuristic, the proportional allocation heuristic, is given in (7.3). It is a minor refinement of the best platform heuristic, allocating tasks inversely proportionally to the makespans of all of the platforms, when all of the tasks have been allocated to each platform. As this heuristic is allocating tasks in proportion to the capabilities of the platforms, in theory it should represent the linear performance improvement discussed in Section 1.2.2.

$$\vec{A}_{i,j} = \left(\vec{L}_i \sum_{o=1}^{\mu} \frac{1}{\vec{L}_o} \right)^{-1} \quad i = 1, 2, \ldots, \mu, j = 1, 2, \ldots, \tau. \tag{7.3}$$

The best platform heuristic performs well when there is a single platform significantly faster than the others. The proportional allocation heuristic is more general, working well provided the elements of $\boldsymbol{\gamma}$ are significantly smaller than the elements of $\boldsymbol{\delta} : \vec{c}^2$ for all platforms. If not, the cumulative constants of all the tasks dominate each platform's makespan, regardless of allocation, and result in a suboptimal allocation of tasks.

Machine Learning-based Allocation

The second approach I propose builds upon the first, using the better of the solutions offered by the two heuristics as a starting point. The platform reduction function $G_L(\vec{F_L}(\boldsymbol{A}, \vec{c}))$ is then specified as the objective function for a time-constrained, global optimisation machine learning algorithm, such as the simulated annealing algorithm provided in SciPy [84], combined with a "polishing", convex optimisation algorithm, such as Danzig's Simplex algorithm, also available in SciPy.

As this machine learning approach incorporates domain specific platform and task information as well as the heuristics, it should at worst confirm the solutions offered by heuristics and at best find the most optimal allocation, achieving super-linear performance scaling, as discussed in Section 1.2.2. As I show in the evaluation in this chapter, a key determinate of the partition optimality is the degree of linearity in the objective function. Furthermore, another factor is problem size, as this problem suffers from the curse of dimensionality with respect to both μ and τ.

I expect that the partitioner such as this would be within the reach of a local developers, as described in Section 2.2.1, however implementing it would require a significant amount of effort and expertise without a certain reward.

Mixed Integer Linear Programming Allocation

Finally, the MILP partitioning approach that I propose uses the formulation of the domain partitioning problem as the input to a constrained integer programming framework, such as SCIP [85], which applies global optimisation techniques as well as a variety of transformations and heuristics to solve the constrained problem.

Frameworks such as SCIP accepts problems in a form very similar to (7.1), however generally do not accept non-linear objective and constraint functions. This requires the problem to be reformulated as given in (7.4), adding additional variables (G_L and $\boldsymbol{B}$) and constraints to capture the non-linearities in the problem.

$$\begin{aligned}
\underset{G_L,\boldsymbol{A},\boldsymbol{B}}{\text{minimise}} \quad & G_L \quad G_L \in \mathbb{R}_+, \boldsymbol{A} \in \mathbb{R}_+^{\mu\times\tau}, \boldsymbol{B} \in \{0,1\}^{\mu\times\tau}, \\
\text{subject to} \quad & \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1, 2, \ldots, \tau, \\
& F_{L,i}(\boldsymbol{A}, \vec{c}) \leq G_L \quad \vec{c} \in \mathbb{R}_+^{\tau}, i = 1, 2, \ldots, \mu, \\
& A_{i,j} \leq B_{i,j} \quad i = 1, 2, \ldots, \mu, j = 1, 2, \ldots, \tau.
\end{aligned} \tag{7.4}$$

Where:

$$\vec{F_L}(\boldsymbol{A}, \vec{c}) = (\boldsymbol{\delta} : \vec{c}^2 \circ \boldsymbol{A} + \boldsymbol{\gamma} \circ \boldsymbol{B}) \cdot \mathbf{1}.$$

As discussed in Section 2.3.3, the MILP approach applies a variety of heuristics to the problem, not dissimilar to the machine learning approach[1]. However, there are two key differences from the machine learning approach: firstly, the approach is able to incorporate the constraints of the problem, and so prune down the solution search space considerably; secondly, the dual formulation of the problem allows for not only a solution to be judged to be provably optimal, but also for meta-heuristics to be applied to assess whether a particular series of allocations is "moving in the right direction". As a result, the MILP approach is far more likely to arrive at super-linear performance improvements, as per Section 1.2.2, as it is better equipped to allocate the best platforms for particular tasks while balancing against the global objective.

However, I expect the use of a MILP partitioner to be out of the reach of almost all domain programmers, barring expertise and experience in management science. Even having such knowledge, the domain programmer would require significant insight in formulating the platform metric models. I would not expect such an approach to accessible, unless provided as part of language/library.

7.1.3. Implementation of Partitioning in F^3

In this section I describe how I have implemented the partitioning approaches I have outlined in the previous subsection within F^3, using the metric models as per the metric models described in Section 6.1.2.

[1]For example, the Simplex algorithm is often used.

Heuristic

Implementing the heuristic solutions, the delta and gamma matrices are generated from the domain metric models implemented within F^3, as described in Section 6.1.3.

The heuristics' formulae, (7.2) and (7.3), are captured in Python functions, as given in Listings A.1 and A.2, that are then used to calculate the heuristic allocation matrices.

Machine Learning

The domain machine learning approach is implemented as follows:

1. The heuristic functions, as described in Listings A.1 and A.2, are evaluated, and the one which has the minimum latency is selected.
2. The shortest latency heuristic is then used as an input to SciPy's simulated annealing algorithm, which has a starting temperature set so as to timeout after 540 seconds, or 90% of ten minutes.
3. SciPy's Simplex algorithm is then used, with the timeout set to 10% of the time spent on the simulated annealing algorithm. This is to "polish" the allocation, in case the allocation has resulted in close to, but not quite an optimum.

Although the performance of Python is generally poor, being a high level, interpreted language, the libraries in SciPy are implemented directly as C or Fortran code [84], and as a result are efficient enough to evaluate a considerable number of solutions in the heuristic value of 10 minutes.

MILP partitioner

The MILP partitioner is implemented in SCIP using the ZIMPL language [85], using the problem form given in (7.4). The Zimpl code for the problem is given in Listing A.3.

As SCIP is a standalone application, it is called from within F^3 using Python's subprocess module, with the parameters for the problem and the resulting allocation communicated via file IO. Similar to the previous section, a timeout of 600 seconds was set.

7.2. Evaluation

In this section I describe my evaluation of the partitioning approaches that make use of domain knowledge, machine learning and MILP. I first characterise the performance of the domain partitioners, the machine learning and MILP partitioners, with respect to problem size and problem non-linearity using synthetic data. I then verify this characterisation using the cluster of real world platforms and tasks described in Section 6.2.1.

7.2.1. Partitioner Characterisation

Synthetic Data Generation Procedure

Drawing upon Braun et al's [40] work, I used the following procedure ($s(\tau, \mu, \theta_\tau, \theta_\mu, \omega_\tau, \omega_\mu, \psi)$) which results in $\boldsymbol{\delta}$ and $\boldsymbol{\gamma}$, the synthetic matrices for characterising the different approaches to partitioning:

1. Construct the baseline vector ($\vec{x}$) and initial matrix ($\boldsymbol{Y}$). $\vec{x}$ is τ uniformly distributed integer elements, bounded by the task heterogeneity factor (θ_τ). $\boldsymbol{Y}$, is a matrix of $\mu \times \tau$ uniformly distributed, non-zero integer elements, bounded by the platform heterogeneity factor (θ_μ):
$$x_j \in [1, \theta_\tau] \quad j = \{1, 2, \ldots, \tau\},$$
$$Y_{i,j} \in [1, \theta_\mu] \quad i = \{1, 2, \ldots, \mu\}, j = \{1, 2, \ldots, \tau\}.$$
2. Construct $\boldsymbol{\delta}$, also a matrix of $\mu \times \tau$ integer values, by multiplying the elements of each row of $\boldsymbol{Y}$ and of $\vec{x}$. i.e.
$$\delta_{i,j} = x_j Y_{i,j} \quad i = \{1, 2, \ldots, \mu\}, j = \{1, 2, \ldots, \tau\}.$$
3. Sort the first $\tau\omega_\tau$ columns of the $\boldsymbol{\delta}$ matrix, and the first $\mu\omega_\mu$ rows, where ω_τ and ω_μ are the degree of task and platform consistency.
4. Construct the $\boldsymbol{\gamma}$ matrix by repeating steps 1-3, however then multiply the resulting matrix by the task constant to coefficient ratio (ψ), i.e. the constant time versus the proportional or splittable time of a task.

Synthetic Case Generation Parameters

The four parameters that I have varied to create the different cases are platform and task heterogeneity, and consistency. The values for what constitutes homogeneous[2] versus heterogeneous[3] parameters are taken from Braun et al's work [40], the authors of which derived it from experimental workloads in a scientific computing HPC centre, hence reflecting real workloads, as would be seen within an empirically defined application domain.

The degree of heterogeneity determines the range over which the values which make up the baseline vector and initial matrices used to generate $\boldsymbol{\delta}$ and $\boldsymbol{\gamma}$. If a partitioning approach is not sensitive to the relative magnitudes of tasks upon platforms, then the heterogeneity factor should not affect the quality of output, whereas an approach that makes assumptions about the relative magnitudes would be.

The degree of consistency determines the strictness of the ordering of the entries in the matrices.

[2] $\theta_\mu = 10$ and $\theta_\tau = 100$

[3] $\theta_\mu = 100$ and $\theta_\tau = 3000$

In the case of task consistency, this determines the ordering of tasks within a platform (the rows of the matrices), i.e. how strictly increasing values will be for a given platform. If a partitioning approach requires tasks to be in a specific order, or indeed the ordering of the task magnitudes is part of the partitioning algorithm, then the task consistency would impact upon the time taken to partition the tasks.

In the case of platform consistency, this is a measure of how correlated task lengths are between platform. A higher degree of platform inconsistency increases the size of the potential for super linear performance improvement, because there is more scope for a partitioning approach to match tasks to platforms according to the comparative capabilities of those platforms[4].

The combination of task and platform consistency reflects the degree of inconsistency between tasks, and hence, the potential for super-linear performance scaling, as described in Section 1.2.

Synthetic Cases

The parameters used in conjunction with the procedure above are provided in Table 7.1. The four cases consider a range of different scenarios, from completely homogeneous, consistent platforms and tasks to heterogeneous platforms running a set of very inconsistent tasks.

Table 7.1.: Synthetic task-platform data generation parameters. Columns are platform heterogeneity (θ_μ) and consistency (ω_μ), and task heterogeneity ($\theta_\mathcal{T}$) and consistency ($\omega_\mathcal{T}$).

Case Designation	θ_μ	ω_μ	$\theta_\mathcal{T}$	$\omega_\mathcal{T}$
Hom-Con	10	1.0	100	1.0
Het-Con	100	1.0	3000	1.0
Het-Mix	100	0.5	3000	0.5
Het-Inc	100	0.0	3000	0.0

Hom-Con The first case considered is homogeneous and consistent platforms and tasks. Such matrices might occur in a scenario where a cluster of very similar platforms is being used to evaluate multiple copies of the same, or very similar tasks.

An example might be a dedicated cluster of identical GPUs performing airflow modelling upon a large, but uniform surface.

Het-Con The second case is platforms and tasks that are heterogeneous, but that are still consistent. Such a scenario could occur with a cluster of platforms that are of the same type, for example multicore CPUs, but are of widely varied compute capacity.

An example would be a cluster of mixed capabilities, including legacy servers being mixed with newer architectures, or web-based, mobile applications with low capability user CPUs co-operating with the IaaS-provided high capability server CPUs. The latter example also provides a rationale for highly heterogeneous workloads, where relatively low requirement compute file

[4]I'm deliberately using the economic terminology, referring to the concept developed by David Ricardo in the 18th century

serving tasks might be mixed with more demanding image manipulation or machine learning tasks.

Het-Mix The third case is again heterogeneous, but with only half of the platform and task entries being consistent. This scenario is possible when different types of platforms of varied ability are mixed together, performing a wide array of tasks.

In this example, in many cases the compute characteristics of the platforms, the clock rate or the degree of task parallelism, determine the run times of tasks, and hence the consistency between platform, but there would also be a significant number of instance where this does not, and platforms have a enhanced capability with regards to a particular task-parameter grouping.

Het-Inc The final case is also heterogeneous, but with no guarantee of consistency. This situation could arise in a scenario when a cluster of extremely heterogeneous, specialised platforms, such as DSPs or ASICs, are being used with a wide array of tasks. As a result, it is difficult to make any assumptions based upon the relative performance of one particular task. Generally, I believe such a situation to be unlikely, but I have included it so as to fully characterise the partitioning approaches.

Synthetic Data Characterisation Results

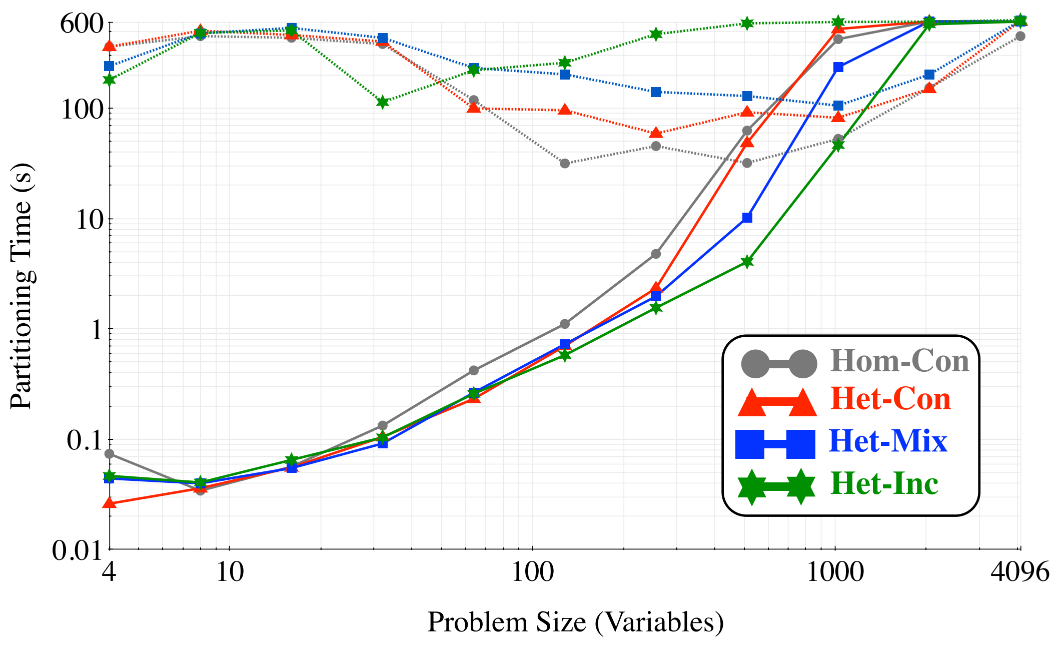
Utilising the synthetic data generation procedure and parameters in Table 7.1, I have evaluated the two domain partitioning approaches in terms of size, i.e. the number of variables in the optimisation problem[5], as well as the ratio between the coefficient and constant latency matrices, ψ. I have reported the time required by the domain partitioning approaches algorithms in Figure 7.1 as well as the quality of the solution returned with respect to the solution returned by the proportional allocation heuristic in Figure 7.2.

Partitioning Time For the partitioning latency experiments, as given in Figure 7.1, a timeout of 10 minutes was set, the same time given to the benchmarking described in the previous chapter. This timeout reflects the amount of time that I believe domain programmers would be willing to spend to obtain some insight into their problem's design space.

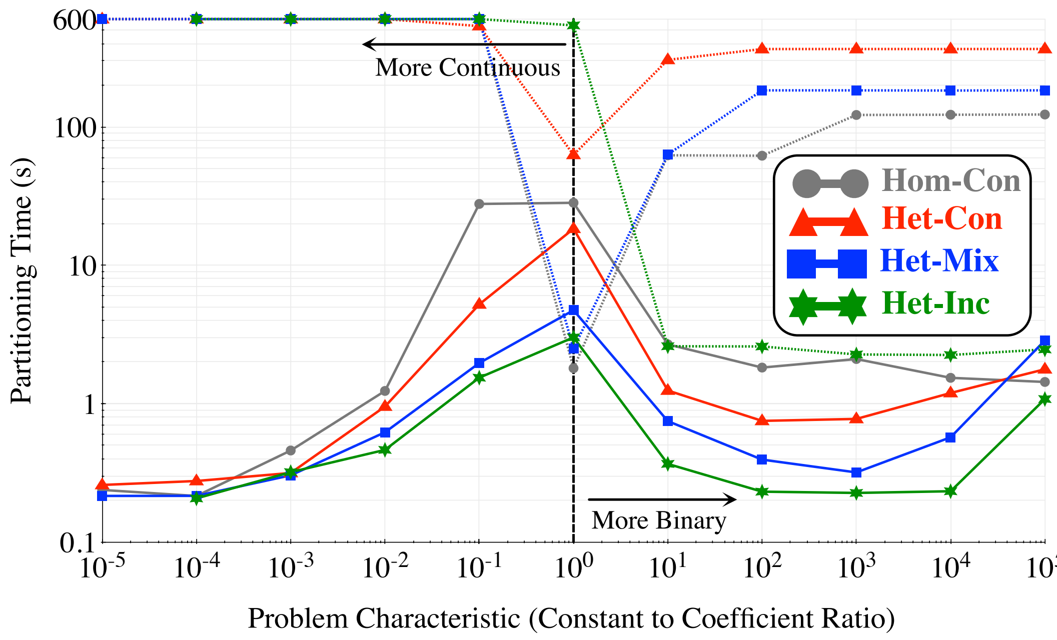
Broadly, as the problem size was varied, as reflected in Figure 7.1a, the machine learning-based partitioner was limited by the timeout, while the MILP partitioner's time grows exponentially as a function of the number of variables. Given the random nature of the simulated annealing algorithm used in the machine learning approach and the non-linear characteristic of the design space, I was not surprised that it takes as much time as given, regardless of problem size.

Further insight is provided when considering when the ratio between the coefficient and constant component is varied, as in Figure 7.1b. The MILP approach's latency peaks around 1, reflecting the considerable linear and non-linear allocation problems that both have to be solved in parallel. This suggests that the scale of the exponential latency seen in Figure 7.1a reflects the worst case for the MILP approach. The machine learning approach performs relatively well

[5] $2\mu\tau$ in this case

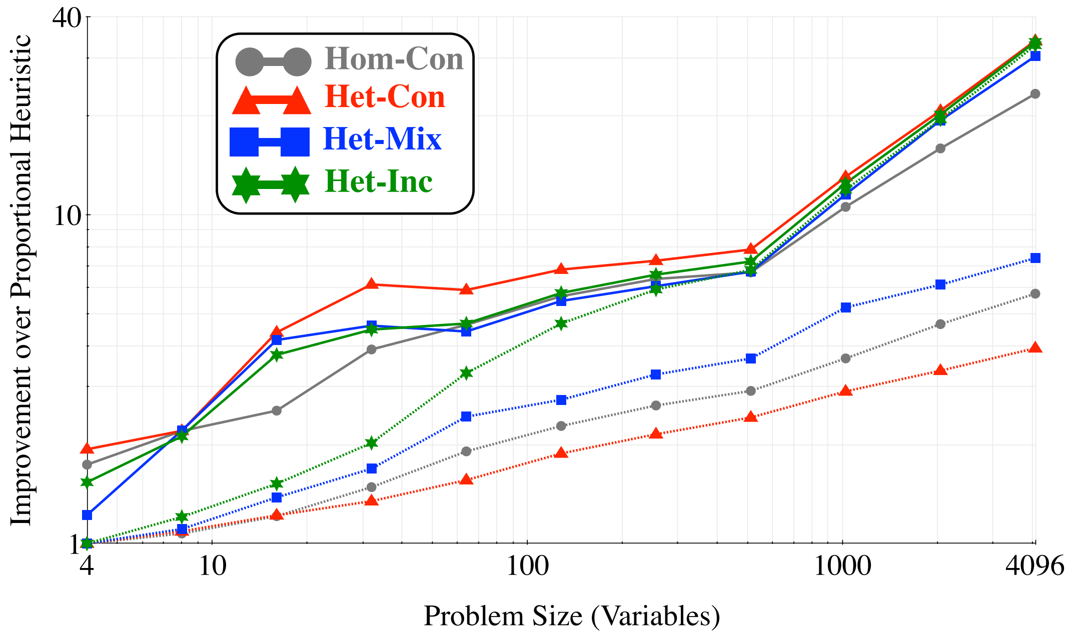


(a) Allocation Problem Size. The constant to coefficient ratio was set to 1.

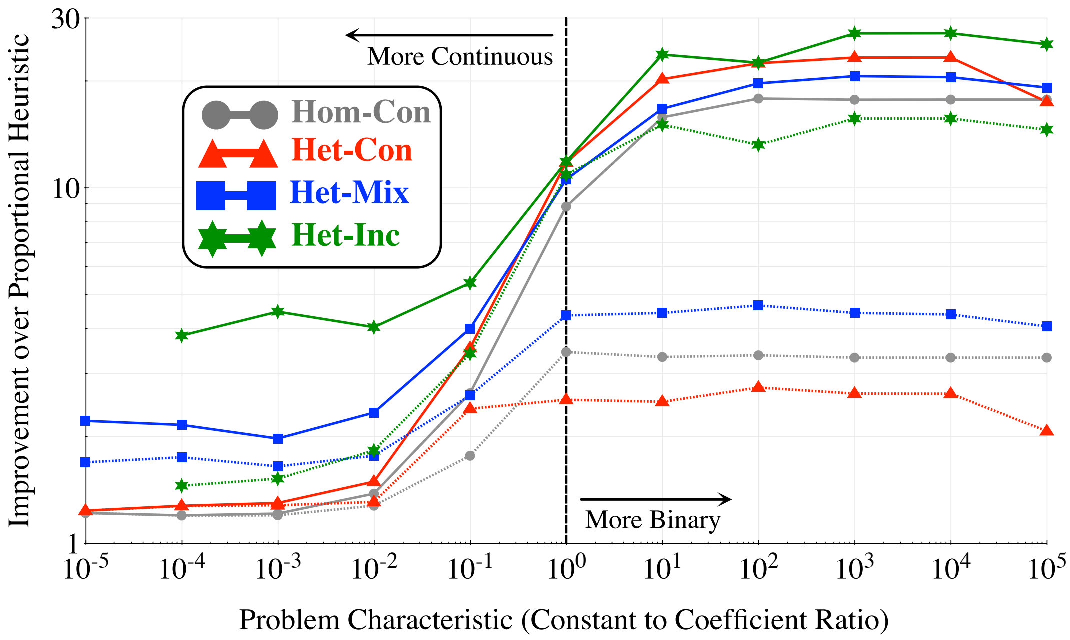


(b) Allocation Problem Linearity. The problem size is 2048 variables.

Figure 7.1.: Latency characterisation of domain partitioning approaches in terms of the time spent partitioning. The dotted line is the domain machine learning partitioner, while the solid is the MILP partitioner. A timeout of 600 seconds was set for both partitioning approaches.



(a) Allocation Problem Size. The constant to coefficient ratio is set to 1.



(b) Allocation Problem Linearity. The problem size is 2048 variables.

Figure 7.2.: Partitioning approach quality characterisation, giving the solution's latency improvement over that returned by the proportional heuristic. The dotted lines are the domain machine learning partitioner, while the solid line is the MILP partitioner.

at this inflection point, suggesting that the heuristic starting point of the algorithm is likely close to the optimal point.

Partition Quality For the quality of the solution relative to the proportional heuristic, Figure 7.2, described in (7.3), the MILP and machine learning partitioners' qualities are expressed as a function of problem variables and constant to coefficient ratio.

As the variable size is varied, as given in Figure 7.2a, I explain that the linear improvement trend is a result of increasing the number of variables, and hence increasing the potential improvement possible over the proportional allocation heuristic, i.e. linear performance improvement.

For the case of the constant to coefficient ratio, as given in Figure 7.2b, the minor improvement seen when there is a dominant proportional component is explained by the proportional heuristic solution being close to the global optimum, hence there not being a large degree of improvement possible. Notably, as the problems become more binary (i.e. more non-linear), the MILP approach show an order of magnitude improvement in all cases, as does the numerical optimiser in the *Het-Inc* case, reflecting that the problems are non-linear, a situation that the heuristics struggle with are being considered.

Discussion The outcome from the quality and partitioning time characterisation is that in what I have defined to be a reasonable amount of time, 10 minutes, both the machine learning and MILP partitioning approaches can find a partition of work better than that found by a heuristic partitioner. This result is certainly testament to the advancement in optimisation algorithms and architectures over the decade since Braun et al's work [40].

The performance of the domain partitioners is explained by two factors: firstly, both approaches take advantage of platform inconsistencies, i.e. the ability of platforms to be better suited to some tasks than others, relative to other platforms. The second factor is how the allocation interacts with the task and platform. For example, by containing feedback loops that reflect the impact of non-linearities in the latency model for example, the domain partitioners are able to account for these non-linearities and hence only allocate tasks to a platform if it is advantageous. By contrast, the heuristic partitioner apply a rule regardless of whether it is advantageousness.

In almost all cases the MILP approach returns the better result, suggesting it should be the preferred method for partitioning workloads. This better performance could be explained by differing quality of implementations in the partitioners, however the scale of the difference suggests that the constrained optimisation process is a better fit for this scenario. Indeed, it is intuitive that an approach that incorporates constraints into its algorithms would be advantageous to one, such as the machine learning approach, which do not.

In the next subsection I address whether this characterisation using synthetic data is borne out when using real platform and task data.

7.2.2. Practical Verification

In addition to the synthetic task-platform data described in the previous section, data from actual pricing tasks and platforms is required to evaluate the different partitioning approaches. This is to confirm that the characterisation performed using the synthetic data is confirmed by what is measured when running an actual workload.

Experimental Workload

For the experimental workload, I used the large set of 128 option pricing tasks from Chapter 6, as given in Table 6.2, as a single workload. 35 of the options in the set are Black-Scholes-based, while the remaining 93 are Heston model-based. A wide range of option types are present in the set, with a bias towards more computationally intensive problems, with most being either Barrier, Double Barrier or Asian option problems. These options were evaluated for a range of accuracy values, from \$1 to \$0.001.

Evaluating such a workload would mirror a real world situation where an analyst in an investment bank might be pricing a large set of options for a portfolio valuation or a Value-at-Risk (VaR) calculation.

As the pricing task parameters are being randomly generated, there is a high degree of variation in the convergence rate of the different option tasks, hence a wide range of values for α in (6.3). Also, due to the random nature of the task generation, there will be inconsistency in the tasks generated for a platform, although as the tasks were generated in batch by type, there will be some ordering, as reflected by the computational intensity of the tasks in Table A.8.

Experimental Platforms

For the platforms, similar to the tasks, I used the platforms from Chapter 6, as described in 6.1.

The variation in platform type, vendor and physical location ensuring a high degree of variability in the execution of tasks in platform execution, and hence a large range of values for β in (6.2). Furthermore, mainly due to the difference in platform type, and hence execution mode, there will be inconsistencies between the platforms.

As the coefficient, i.e. δ in (6.4) is defined as $\delta = \beta\alpha^2$, both the platform and task heterogeneity of the real world data is high. Furthermore, the tasks and platforms, while not completely consistent, as the results in Tables A.10, A.11 and A.12 reflect, are also not completely inconsistent, as illustrated by the mean performance results. Hence, the real world data most resembles the *Het-Mix* case in synthetic data in Section 7.2.1.

Practical Verification Results

While the characterisation using synthetic data of the domain partitioning approaches provide insight into the domain partitioning approaches, I have verified these results with real platform and derivatives pricing domain task data. I put the portfolio of pricing tasks in Table 6.2 through the partitioning approaches for the platforms in Table 6.1 for a range of accuracies. I then ran the generated partitions, and measured the domain metrics of latency and accuracy.

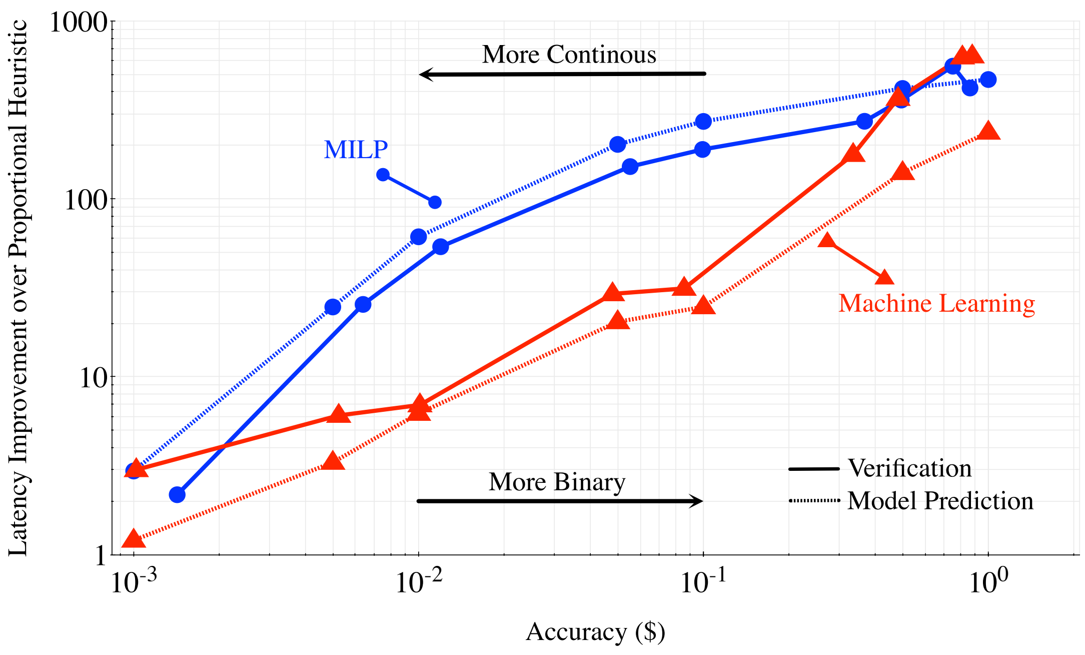


Figure 7.3.: Improvement of Numerical Optimiser and MILP Partitioners over proportional heuristic for domain models results and verification. The model is the dashed line, while the verification data is the solid line.

Synthetic vs Real data Figure 7.3 is the solution improvement over the proportional heuristic using platform task data from the domain metric models and the verification of the generated partitions.

The shapes of the improvement curves, similar to that seen in Figure 7.2b, shows that the characteristics of the actual and synthetic data in the previous subsection are similar, although there is an order of magnitude difference between the two. This is explained by the fact that while the synthetic data generation procedure approximates the real world scenario, it is still drawn from a uniform random distribution, while the real world data is almost certainly from a different distribution, defined by the nature of the platforms and tasks. The domain metric models and the metrics measured at run-time are generally close in value, within the 10% extrapolation error demonstrated in Section 6.2.2.

Another significant feature of the results is the dramatic improvement over the proportional heuristic seen at lower accuracies (larger 95% confidence interval values), i.e. when the problem is more binary. This is due to the proportional heuristic performing very poorly when the latencies of platforms are mostly composed of the constant, setup time. There is considerable scope for improvement here, and hence both domain partitioning approaches perform extremely well. When the problem is more continuous, i.e. at the higher accuracies, the improvement is much less, where the improvement is solely due to the task-platform matching effect.

Pareto Curves Figure 7.4 illustrates that the projected values for partitioning approaches are close to what is measured in reality when the allocations are run. The differences between the predictions and what was measured when executed are well within the 10% error of the domain metric models. Furthermore, both the domain knowledge-based machine learning and MILP-

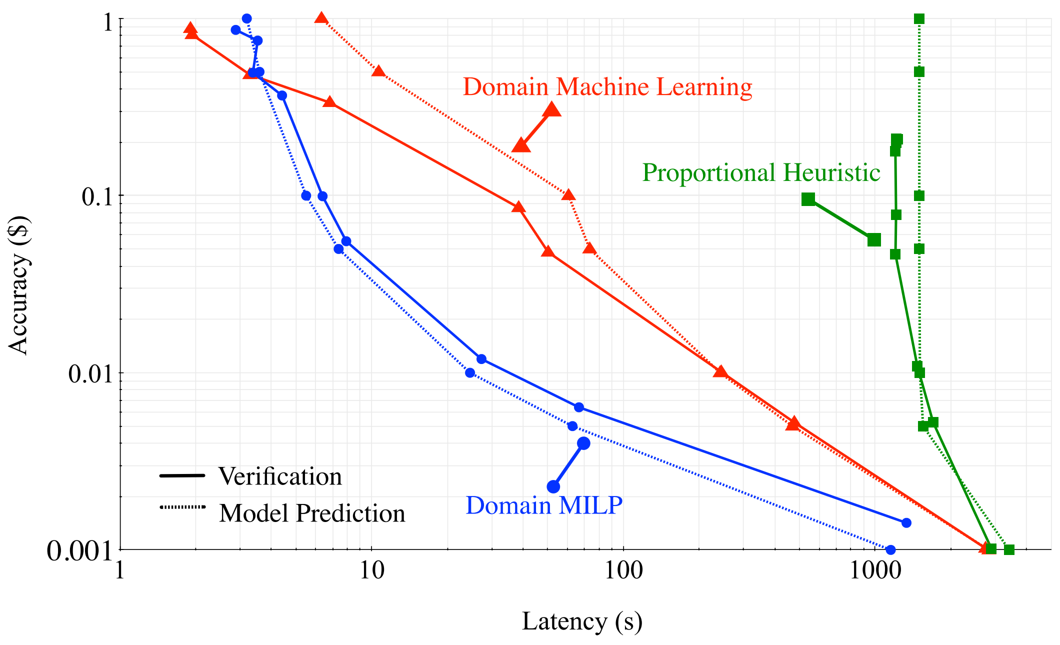


Figure 7.4.: Resulting Pareto curves from different partitioning approaches. Dotted lines are the partitioner metric predictions, while the solid lines are metrics values when executed on experimental platforms.

based partitioner are orders of magnitude more efficient than that suggested by the proportional heuristic for problems with strong non-linear characteristics, in this case, when accuracies greater than \$0.005 were required.

The Pareto curves in Figure7.4 illustrate the considerable difference in metric values that result simply from changing the allocation of tasks to platforms. The crucial difference between the approaches is how each copes with the nature of the tasks. At the lower accuracy (i.e. larger 95% confidence interval values), the domain MILP and machine learning approaches use fewer platforms that have a low network latency, and hence low constant component cost. At higher accuracies, more platforms are used, regardless of the networking time, as it constitutes a very small component of the tasks' latencies in general. By contrast, the proportional heuristic does not cope with non-linear problem latencies well, and hence uses all platforms for all tasks, even when the cost of doing so far outweighs the gain.

While the allocations of work produced by the domain partitioners for the real workloads in the specified timeout were not provably optimal, I did find that truly optimal partitions could be found, if the partitioners were run for long enough[6]. While the performance of these approaches relative to competing approaches is of interest for practical interest, the close to theoretical optimality of the results strengthens the argument for this approach.

The results from applying the partitioning approaches to real domain task and platforms verify the results of the evaluation in the previous subsection that used synthetic data. This evaluation also represents the culmination of the domain specific methodology outlined in Chapter 3, producing a domain metric representation of the heterogeneous computing design space in Figure

[6]Often multiple hours, hence the need to set a timeout that would be realistic for a domain programmer

7.4 automatically. A domain programmer would intuitively understand this representation, and hence it would help them use heterogeneous computers more effectively.

7.3. Partitioning Heterogeneous Platforms

In this chapter I have demonstrated how the domain specific methodology can be used to partition tasks across heterogeneous computing platforms, as asserted in Section 3.3. I described three different approaches to partitioning that can take advantage of the information provided by the domain metric models:

1. Two heuristic partitioners based upon naive rules, the best platform and proportional allocation heuristics.
2. A machine learning partitioner based upon simulated annealing and the Simplex algorithm
3. A MILP partitioner using Branch and Bound, the Simplex algorithm and other optimisation techniques.

While none of the approaches described require domain metric models, as relative platform performance, as reported in vendor datasheets[7], or application benchmarks, could be used to calculate allocations. However both the machine learning and MILP partitioners demonstrate significant performance improvement over the heuristics if the information from domain metric models is used, hence I identified both as domain partitioners.

However, I acknowledge that the domain partitioning described and evaluated in this chapter is a limited achievement of the vision of automated support for heterogeneous computing described by Braun [38] and illustrated in Figure 2.4. This is because only the partitioning of independent tasks is considered, with the makespan of all tasks being the only latency consideration. A fuller approach would be capable of scheduling, both in terms of absolute latency requirements of each task, as well as the dependencies between tasks. Hence this approach is only half of a full mapping process. However, as has been motivated by me in Section 7.2.2, and in the literature [40, 45], scenarios such as this do occur, and hence this is a contribution.

This chapter concludes the case study of the derivatives pricing domain, and my practical demonstration and evaluation of the domain specific methodology described in Chapter 3. In my case study of the domain of derivatives pricing, I have been able to demonstrate the three features of portability, prediction and partitioning. In addition to proving each of these features in isolation, I have shown how each successive feature builds upon the previous one, serving as a demonstration of both the previous feature's validity and viability.

In the next Part, I resume my discussion of the general domain specific methodology, in light of the results from the case study, as well as consider the limitations of the methodology. I then conclude this dissertation, reflecting upon how I have addressed the research problems outlined in the Introduction.

[7]With the usual caveat about claims made by vendors about their products

Part III.

The Analysis

8. Discussion

In this final part of my dissertation, I analyse my proposed domain specific methodology in light of the derivatives pricing case study, as described in the previous four chapters, and more generally. In this chapter, I discuss the degree to which the the case study supports the domain specific methodology for heterogeneous computing I have proposed, and the limitations of both the case study and methodology. In the next chapter, which concludes the dissertation, I consider the degree to which the research questions raised in the Introduction have been answered as well as future directions for this work.

I now reflect critically upon the claims that I have made in Chapter 3, and throughout Part II. I have argued that a using domain specific approach with heterogeneous computing platforms enables three features:

1. Portable, efficient implementation of a single, high level task description upon a diverse set of heterogeneous computing architectures.

2. Prediction of the run-time characteristics of a task upon a heterogeneous platform.

3. Partitioning of multiple tasks across many heterogeneous platforms so as to balance run-time characteristics.

As this is an engineering dissertation, I am not only interested in proving that these three features of a domain specific approach exist, but are also practically realisable. To this end, to both prove the existence and practicality of the features, I considered a case study of the application domain of derivatives pricing. This case study also serves as an extended explanation, by way of demonstration, of the domain specific methodology for heterogeneous computing that I described in Chapter 3. In order for my claim to hold that a domain specific methodology enables the three features outlined above practically, I need to successfully argue that the case study demonstrates these three features in practice, and show that the case study is a valid instance of the methodology.

Furthermore, I delineate the limits of this methodology by expanding upon the assumptions that it makes, and beginning to address its shortcomings.

In the first section of this chapter, I consider the case study with respect to the domain specific methodology. I first consider the achievement of the three features according to the criteria identified earlier. In the second, I address the methodology itself, and its limitations with respect to underlying assumptions. In doing so, I propose remedies to these limitations, that are not necessary to validate the methodology, however enhance what is being proposed.

8.1. Does the Case Study validate the Methodology?

In this section I evaluate the relationship between the case study, the domain specific methodology and my broader thesis. I do so by first considering the degree to which the three domain specific features are implemented, and I then reflect upon the limitations of the case study.

8.1.1. The Case Study as an instance of the Methodology

In Sections 3.1.4, 3.2.4 and 3.3.4, I laid out the criteria for the achievement of each feature of my domain specific methodology for heterogeneous computing. In this subsection, I reflect upon the degree to which each feature has been achieved.

Portable, Efficient Execution

In the first instance, the criteria for the portability feature is the existence of a domain specific means to implement a single task description upon multiple, heterogeneous computing platforms. The portability feature was demonstrated in the case study in Chapter 5 through F^3, which accepts tasks described using a library of Python objects, and is capable of implementing these tasks upon a range of multicore CPU, GPU and FPGA platforms, as evidenced by the diversity of the experimental platforms supported, as described in Tables **??** and A.5.

Evidence of the efficiency of these implementations is given in Sections 5.2.2 and 5.2.3. Further evidence is given in Table 8.1, where I have detailed the performance of the experimental platforms from Section 5.2.1 with respect to an optimised, but sequential, CPU POSIX implementation. Almost all of the platforms exhibit acceleration over the sequential implementation on the Desktop platform, despite running at the same or lower clockrates[1]. Such acceleration is only possible if the implementations achieved through the domain specific methodology are efficiently portable, i.e. the unique computational resources of the diverse platforms are being exploited.

A further point to note is the degree of inconsistency between platforms and tasks. There is significant variation in performance of different categories of tasks, i.e. within the rows of Table 8.1, suggesting that there are differences between how these tasks are executing on the different platforms. Furthermore, the performance of any one task on a platform is not necessarily a predictor of performance for another task on that platform. It is this inconsistency phenomena that makes the prediction of domain performance metrics as outlined in Section 3.2, and

demonstrated in Chapter 6 necessary, and super linear performance scaling described in Section 1.2.2 achievable through the domain partitioning outlined in Section 3.2.4 and as demonstrated in Chapter 7.

Predictive Domain Metric Modelling

Two criteria were defined for the domain metric modelling feature in Section 3.2.4. Both of these criteria are premised upon the relative predictive error of the models, as given in (3.8) - the difference between the metric values predicted by the model for a particular instances

[1]Despite many of the platforms being newer than the sequential CU used, reflecting that clock-rates are now relatively static

Table 8.1.: Relative latency of F^3 Implementations with respect to sequential CPU implementation for Kaiserslautern Heston model benchmark options [81] and Imperial College London's Black-Scholes model Asian option [80].

Designation	H-E	H-B	H-DB	H-DDB	BS-A	Mean	Variance
Sequential CPU			1.0			1.0	0.0
Desktop	4.13	3.92	3.79	4.34	4.06	4.04	0.04
Server	11.03	10.97	11.79	11.47	9.98	11.03	0.47
Manycore	25.61	24.29	25.28	25.57	21.68	24.44	2.75
NVIDIA Workstation	482.51	285.67	269.19	281.83	233.15	300.20	9,679.34
NVIDIA Cloud	941.47	558.77	526.87	551.05	454.14	586.44	36,777.48
AMD Workstation	718.51	304.72	263.90	276.47	378.97	360.08	36,023.48
Phi	2093.14	589.21	447.63	444.87	1119.23	772.38	493,120.85
ZC706	6.85	3.38	3.14	3.26	3.37	3.81	2.55
PCIe-A7	158.95	109.11	92.29	102.59	67.21	101.99	1,129.44
PCIe-D5	420.09	225.99	133.51	148.11	279.25	220.75	13,486.46
Max3	178.67	111.48	103.63	107.42	149.58	127.10	1,075.07
Max4	263.59	165.56	154.84	159.30	216.09	187.64	2,213.74

of the implementation variables, and the metrics measured when the task is run with those implementation variable values.

The first criteria, *incorporation*, is that the models become increasingly accurate when provided with additional information. This criteria is necessary, but not sufficient for the existence of a useful model. The second, extrapolation, is that the error in predictions remains sufficiently bounded for a large set of implementation variables, which I heuristically define to be 10% for over an order of magnitude change in the implementation variables.

The evaluation undertaken in Sections 6.2.2 and 6.2.3 for the metric models described in Chapter 6 reflects that these criteria hold for the latency and accuracy metric models for the pricing function in the derivatives pricing domain. The evaluation was performed using a varied set of tasks and heterogeneous platforms, suggesting that these criteria are holding across a broad range of task and platform types.

Figure 8.1 synthesises the results of the derivatives pricing metric modelling into the unified latency-accuracy model described in (6.4), illustrating how the domain specific approach enables the abstraction of heterogeneous platforms using the metrics of the application domain. The trade-off curves are a representation of the domain design space, as described in Section 3.2.1, for Table 6.2's pricing tasks on Table 6.1's platforms, achieved by varying implementation variables. The latency at each accuracy point is the sum of all of the projected latencies for the tasks, i.e. the makespan of the set of tasks, at that accuracy level, as per the unified metric model for each task.

As is to be expected, with the lower accuracy requirement, and hence smaller number of paths required, the latency ordering of platforms is determined by the constant setup time. Of the constant component, the platforms' network latencies are generally the largest component, hence local platforms have shorter makespans. However, as the accuracy requirement increases,

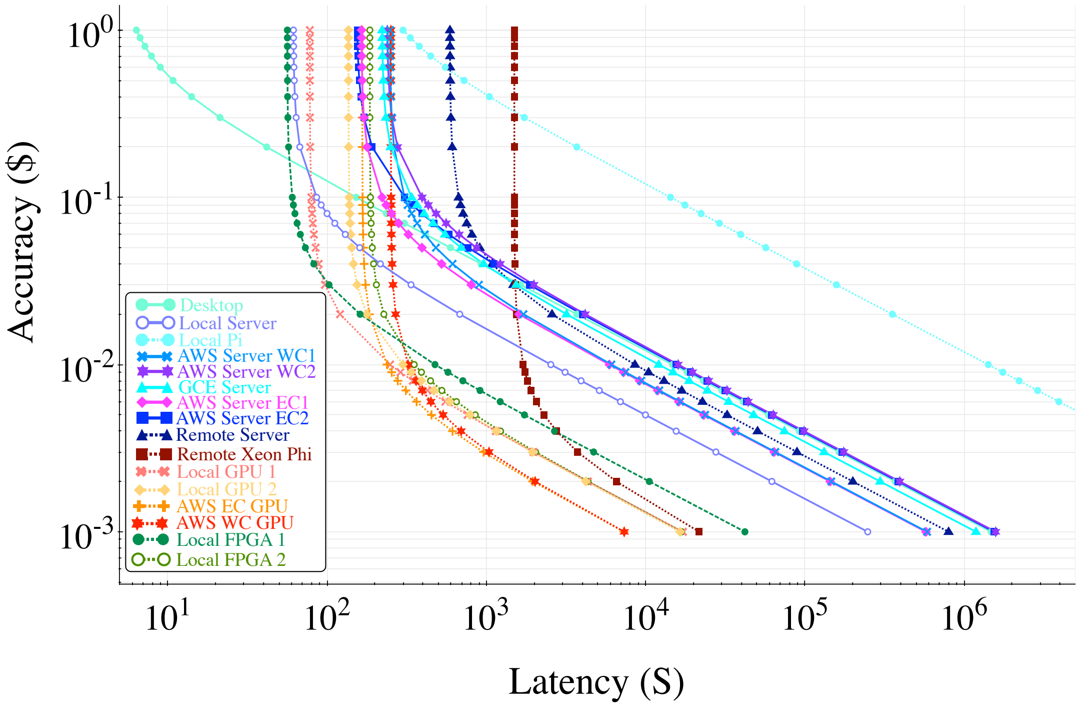


Figure 8.1.: Pricing function latency-accuracy metric trade-off curves of individual platforms for the 128 option pricing problems given in A.8

the ordering is determined by the relative computational capabilities of the platforms, hence the GPU and FPGA platforms are an order of magnitude better than the CPU platforms.

This illustration of the design space, with the platform crossover points, suggests the next logical step: using multiple heterogeneous platforms to evaluate multiple tasks, based upon the insights provided by the domain metric models, especially if this allocation approach is going to exploit the inconsistencies exhibited in Table 8.1.

Partitioning of Workloads

Three criteria were identified in Section 3.3.4 for the automated, optimal partitioning that I claimed the domain specific approach enables in Section 3.3.1 and demonstrated in Chapter 7. The first two criteria are practical - that the partitioner provides an allocation that performs all of the specified tasks[2], and that the resources, be it time or otherwise, that the partitioner requires does not dwarf the resources used by the resulting partition. The final criteria was broader - that the partitioner finds the Pareto optimal, or close to Pareto optimal, allocation of tasks to platforms for a range of metric values.

In Section 7.1.2, I described three different instances of the approaches outlined in Section 2.3.3 for partitioning that can make use of derivatives pricing domain knowledge. The heuristic approaches use simple algorithmic rules, and are hence easy to compute. Due to the simplicity of the rules however, these heuristics cannot exploit platform and task inconsistencies. The second and third approaches, which I refer to as the domain partitioning approaches, are responsive to the domain metric models, and hence platform and task inconsistencies. The second partitioning

[2]As the domain programmer would assume

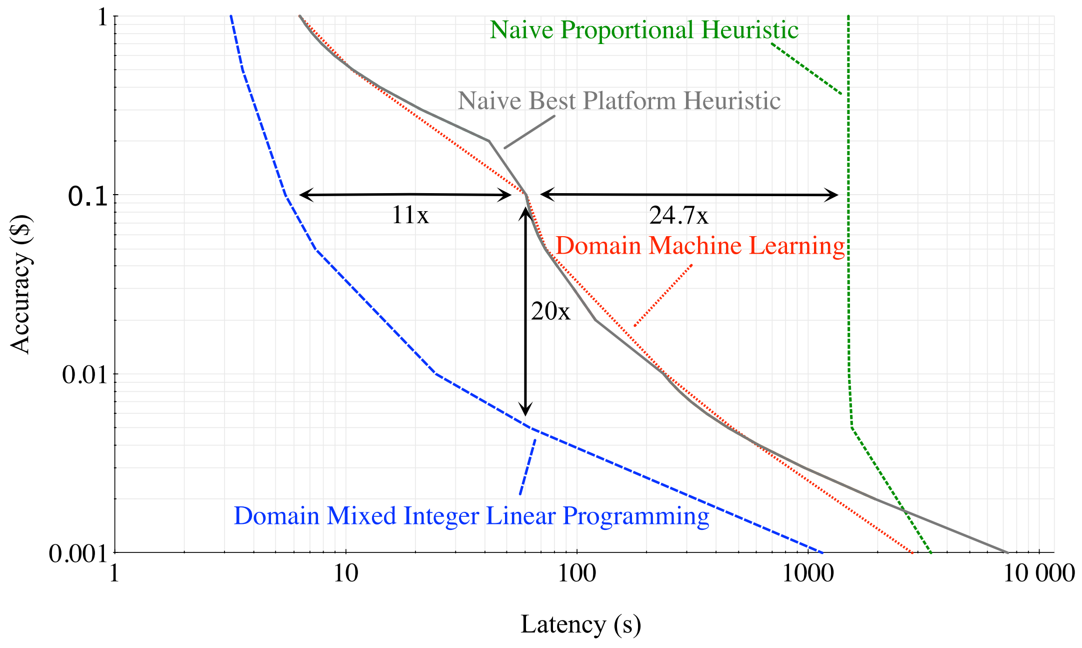


Figure 8.2.: Comparison of efficacy of different partitioning approaches.

approach uses global optimisation, machine learning algorithms, such as simulated annealing and the Simplex, whilst the third uses MILP techniques.

As the experiments undertaken in Section 7.2.1 describe, the domain partitioning approaches are both practical, achieving good results in what I have heuristically defined is a reasonable amount of time to the domain programmer. The domain approaches are also more optimal than the simple heuristics, in some cases improving the metric being optimised by more than an order of magnitude, as illustrated for both latency and accuracy in Figure 8.2.

If Figure 8.1 is the domain specific representation of the design space for each of the platforms, Figure 8.2 is the combined design space for those platforms and tasks. It is a complete abstraction of the heterogeneous computing resources that a domain programmer would be able to intuitively understand, and hence use to balance their requirements in a way that an automated effort could not.

Furthermore, each of these features have been shown in practice to build upon the previous, and hence helps serve as a demonstration of the utility of the proceeding feature. The financial task partitioning in Chapter 7 would not be possible without the latency and accuracy metric models in Chapter 6, which in turn would not be possible without the portable implementations developed in Chapter 5.

8.1.2. The limits of the case study

In this subsection I consider the limitations of the derivatives pricing case study. Throughout the case study, I provided explanations for the experimental methodology and results analysed, however if the criticisms below hold, then it is possible that the results of the case study are valid, but are insufficient to support the claims of the domain specific methodology.

I consider two potential challenges to the case study in the context of this project: The first challenge is that the case study is too narrow to be considered a full evaluation of the domain specific methodology I proposed in Chapter 3. I consider this challenge in terms of the three principal features of the methodology. The second is that this case study is exceptional, and hence not representative of most application domains considered.

Narrowness

A potential criticism of my case study is its narrowness. Not only is the domain considered limited, but within this area, I only consider two performance metrics and a limited number of instances of the partitioning approaches. Below, I address these critiques:

The derivatives pricing domain: The first narrowness critique is that the derivatives pricing domain itself is narrow, by definition only having one domain function, the pricing of derivatives. I will add to this critique, observing that I only consider one algorithm for performing the pricing domain function, where multiple exist.

The benefit of domain specific abstractions in the context of heterogeneous computing has been demonstrated by other researchers, as described in Section 2.2.3. Hence I am rather confirming the claim as opposed to originating it. I focused my efforts on supporting a wide array of platforms and programming standards, as I observe that for the other work such as Chafi et al [4], supporting larger domains appears to have come at the cost of only supporting one or two heterogeneous computing technologies, connected by at most one communication technology.

I justify the sole use of the Monte Carlo algorithm in light of its computational robustness - it grows linearly with the underlying model complexity as opposed to the exponential growth that other algorithms such as finite difference methods exhibit [74]. Where I believe that I have demonstrated diversity is in the multiple underlying models, both the Black Scholes and Heston model, and the many variations on option products that I considered in my evaluations.

Only two domain metrics: A further scope critique is that I have only considered two domain metrics - latency and accuracy. I address this by suggesting that I went further than the burden of proof that was actually required.

In Section 3.2, I motivated the domain metric models as domain specific abstractions of a particular task implementations upon a heterogeneous platform. Hence, I only needed to demonstrate for one domain metric that such a model exists. I chose to implement two metrics however, because in relating the two models together, as I did in Section 6.1.2, I could show the potential for design trade-offs to be presented in domain specific terms.

A further motive for the two metrics is that in my reading of the computational finance literature, I found that these two metrics are of the greatest concern to financial engineers. I could have added additional metrics, for example the financial cost[3] of a particular task, or additional statistical properties of the derivative product's price distribution. However many of these metrics are largely a function of the two already considered, in the case of cost, it would largely be a function of latency, hence adding this model wouldn't have enhanced the study further.

[3]Often more a concern of the banks' CTOs than the financial engineers themselves.

However for completeness sake, I have provided a cost model in (8.1). The metric model for the task cost on a particular platform ($f_S(f_L(n))$) is given by rounding up the latency of the task divided by the cost quantum (ρ), and then multiplying by the platform rate (π). The platform rate could be provided by the IaaS provider that is providing the platform, or using a cost model, such as the Simple Model for data centre costing from the Uptime Institute [86], that incorporates the device energy cost use as well as the capital cost of the device itself.

$$f_S(f_L(n)) = \left\lceil \frac{f_L(n)}{\rho} \right\rceil \pi \tag{8.1}$$

Limited number of heuristics: The final potential critique pertaining to the task narrowness is that I only consider two partitioning heuristics, as described in Section 7.1.2, given that a wide variety of heuristic approaches are generally employed in the field of distributed computing, as detailed in Braun et al's work [40].

I chose two heuristics to reflect "common sense". In the first heuristic, by allocating work to the single, best platform available, for the second, all platforms are used, in proportion to the relative capabilities of the various platforms. Both reflect what I believe a reasonable domain programmer would implement, given a limited amount of time. Even more advanced heuristics however, as described in Section 2.3.3, are still limited, particularly in the presence of platform and task inconsistency.

The exceptionality of derivatives pricing

It could be argued that the derivatives pricing domain, and the Monte Carlo algorithm used is particularly well suited to implementation upon heterogeneous computing platforms. To elaborate further, by virtue of being compute bound, and "embarrassingly parallel", as described in Section 4.2.2, it is particularly suited to the parallel architectures such as the multicore CPU, GPU and FPGA platforms considered in my experimental evaluations. There are two criticisms here, firstly F^3's implementations could be dramatically inefficient, and that secondly the domain itself is a "soft" target.

The true efficiency of F^3: The first critique has been addressed in section 5.2.3, in the inter-platform performance results reported, F^3 compares favourably to expert implementations from the literature [80, 81]. I will concede however that an implementation created by a platform expert, employing both algorithmic as well as hardware knowledge, would probably achieve better performance than F^3on any of the platforms surveyed.

Firstly, F^3 is using the full compute capability of the target hardware, as evidenced by the results in Section 5.2.2, suggesting that while it might not be completely optimal, it is using the full capabilities of the platforms being targeted.

However, truly optimal implementations are not the point, which is rather that F^3 provides the means to use the heterogeneous platforms where there would otherwise be no capability. To phrase this in resource trade-off terms, while F^3 might be more inefficient compared to an expert implementation, it is likely more efficient in producing that implementation as opposed to cost of employing said expert.

Cherry Picked Domains: I will concede though that the derivatives pricing domain is particularly well suited to parallel execution.

This is a moot point however, as the domain has utility beyond being being a computational exercise. If I had used a domain particularly ill-suited to parallel execution, say one based upon a finite state machine abstraction[4], that would not invalidate the methodology, but merely provide a rationale for the relative performance of implementations across the platforms supported.

8.2. What's wrong with the Methodology?

In this section I consider the limits of the domain specific methodology which I propose in Chapter 3 more broadly. I do so in terms of the implementation effort the methodology entails, the assumptions made with respect to application domains, and finally the expansion of the partitioning feature to incorporate scheduling, and hence achieve mapping, as described in Section 2.3.2.

8.2.1. Upfront Implementation Effort

Fowler and Parsons, as well as Mernik et al [2, 3] make the first criticism of the domain specific development approach. Their critique is that the approach entails significant development effort before a useful domain specific program is written[5]. Not only does the necessary supporting infrastructure need to be implemented, but even more costly is the expertise and experience that will need to be brought to bear in the analysis of the domain and platforms being targeted, and hence the structuring of the domain specific means of expression and compilation framework.

The same authors suggest that this large cost can be diminished by pursuing an iterative process, i.e. first implementing an application framework which is more easily modified and changed, and only moving to a full domain specific language when the structure and utility of the domain functions have been identified. A further advantage to this approach is that it also offers the opportunity to identify the most commonly used domain functions, and hence those that need to be supported upon heterogeneous platforms.

This remedy however only amortises the development cost over a longer period of time. It is a fundamental limitation of the domain specific methodology that I propose, as all forms of abstraction of computing, that it requires insight and effort up front. It is up to the stakeholders identified in Section 2.2.1 to balance this cost against the features of the approach that I have demonstrated, and decide whether it is worth it.

To attempt to quantify this development effort versus efficiency trade-off, I draw upon the results from the literature from Table 2.2. These results suggest that the productivity benefits of a domain specific approach is roughly a 4 times improvement, hence if more than 4 times effort will be spent in using the domain specific system as opposed to created it, then there will be a net positive productivity benefit. This assumes such development effort can be accurately predicted.

[4]Sometimes referred to as "embarrassingly sequential"

[5]To say nothing of the risk associated with the development of any complex system

8.2.2. Assumption of Domains

The second criticism of the methodology that I consider is that it assumes applications domains have the two properties which the features I have outlined require. The first property is the existence of the disproportionately useful domain functions, and the second is that the domain metric models will be deterministic in nature.

Distribution of Function Use

I argue that portable, efficient implementations are possible due to the presence of a power law distribution of domain function use. I rely upon there being a small number of domain functions that are used far more frequently than others, hence the system programmer can focus on implementing these functions efficiently upon heterogeneous platforms. The domain programmers will then be effectively able to make use of these heterogeneous platforms by virtue of using these critical domain functions.

While I have found some empirical research to support this claim in the end user programming literature [28, 29], I will concede that there is limited support. I do however argue that the existence of application domains and specialised architectures at all suggests that a skewed distribution of function use does exist. All of the functions within a particular domain are those that are used infinitely more than those not included in the domain, and so it is not such a stretch to suggest that the motivation for identifying an application domain is based upon a core set of commonly shared functional requirements, and a larger set of supporting requirements.

It is possible that there is need for a taxonomy of application domains, with "functional domains" being defined as application domains that are grouped around a single or a small group of functions. I suggest that the derivatives pricing domain would be a good candidate for being identified as a functional domain.

Determinism

The second assumption made with regards to application domains by my approach is that deterministic models that map implementation variables to domain metrics can be found for all functions. However, as I have demonstrated in the derivatives pricing case study, the only requirements I have of the metric models that are probabilistic in nature. The metric models hence only require correlation between the implementation variables and the domain metrics of interest.

8.2.3. Scheduling

The final criticism of my domain specific methodology is of the third feature, partitioning. As described in Section 8.1.1, I argue that this feature has been implemented, however I do note that it does not completely fulfil the vision of automated support for heterogeneous computing laid out in Braun et al's work [38], and described in Section 2.3.2.

To provide the *mapping* of tasks to platforms, the approach needs to not only partition tasks across platforms, but also schedule the tasks upon those platforms.

The simplest way to cope with scheduling is by evaluating many partitioning problems, based upon the evaluation of the levels of a directed acyclic forest of graph trees representing task dependencies. The practical problem with such an approach is firstly the considerable computational complexity of the partitioning approaches, as characterised in Section 7.2.1. However, the potentially bigger problem is that any task that has less dependencies than any other task might become a bottleneck for all other tasks.

Alternatively, the approach given in (2.1) can be expanded to cope with scheduling by introducing an additional task scheduling vector variable to the allocation matrix $\vec{B}$, and an additional set of constraints that capture the dependencies between tasks, as given in (8.2) for a case when a single domain metric, F_k, is being considered, and where task w depends upon task v.

$$\begin{aligned} \underset{\boldsymbol{A}\in\{0,1\}^{\mu\times\tau},\vec{B}\in\mathbb{R}^{\tau}_{+}}{\text{optimise}} \quad & F_k(\boldsymbol{A}), \\ \text{subject to} \quad & \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1,2,\ldots,\tau, \\ & C_v \leq B_w \quad \vec{C} \in \mathbb{R}^{\tau}_{+}, v, w \in 1,2,\ldots,\tau,\ v \neq w. \end{aligned} \tag{8.2}$$

Where:

$$\vec{C} = \vec{G_L}(\boldsymbol{A},\boldsymbol{B}).$$

The expanded latency task reduction function ($\vec{G_L}(\boldsymbol{A},\boldsymbol{B})$) returns the vector of the upper latency bound for each task ($\vec{C}$), for the given allocation and scheduling variables. As scheduling inherently requires prediction of task lengths, a latency prediction model is required. This formulation represents the mapping problem as an optimisation problem, however it does require a prediction of the runtime of each task upon each platform, which might not be available for a particular platform or task. However, it is suitably general that it may be applied in many situations.

8.3. Concluding the Discussion

In this chapter I resumed the discussion of the domain specific methodology, reflecting upon both the results of the case study in Part II, and in more general terms. I first considered the relationship between the case study and the methodology: did the case study validate the methodology, and the weaknesses of the case study itself. I then considered the limitations of the broader methodology.

I showed how the each of the three features of the domain specific methodology are not only present in the case study, but are also valid. I also addressed concerns around the limited subset of data types and functions supported, domain metrics considered and partitioners used.

However, I did concede that the methodology requires significant upfront development effort, and does not outline a full mapping approach. I proposed two remedies to the second point, showing how scheduling could be achieved in addition to partitioning.

In the next chapter I conclude this dissertation, showing how the research questions posed in the Introduction have been addressed, as well as suggesting future directions for this work.

9. Conclusion

This chapter concludes this dissertation on a domain specific methodology for heterogeneous computing. In Chapter 1.1 and Part I, I argued that domain specificity enables three key features in the context of heterogeneous computing, Portability, Predication and Partitioning, and described a methodology for realising these features. To both illustrate my argument, and evaluate it, I undertook a case study of the domain of derivatives pricing in Part II. In Chapter 8, I refined my argument from Part I, in light of the results of the case study, as well as potential critiques of both the case study and the methodology in general.

I now reflect upon the degree to which I have addressed the research problem that I outlined in Section 1.3. In order to do so, I need to consider whether I have sufficiently answered the research questions raised. I also need to consider the future work of this project, both as a guide to future researchers, but also as confirmation of the positive findings of my work, as evidence of its value.

I first consider the degree to which the research questions outlined in Section 1.4 have been addressed. I then suggest various extensions to this work, and explain why they would enhance and extend the claims that I have made. Finally, I conclude by revisiting the motivation for this project, and reflect upon the relationship between the programmer and the computing hardware.

9.1. Have the Research Questions been answered?

In Section 1.4, I posed the following three research questions:

1. Is it feasible to support the portable, efficient execution of a single computational task description upon a heterogeneous computing systems?
2. Can the run-time characteristics of tasks be predicted across heterogeneous computing systems?
3. Is it possible for tasks to be partitioned across heterogeneous computing systems so as to balance programmer objectives?

In Section 8.1, I argued that the derivatives pricing case study validated the domain specific methodology. I now consider the degree to which the domain specific methodology answered these questions, and the qualifications that these answers inevitably bring.

9.1.1. Portable, Efficient Execution

The first question has been answered, both in the worthy work of other researchers as described in Section 2.2.3, but also in my methodology in Section 3.1 and the case study in Chapter 5.

The experimental verification I undertook demonstrated that my derivatives pricing framework, F^3, is making good use of heterogeneous, parallel computing resources. I have verified both relative to each platform's theoretical performance, as well as to external, programmer created implementations, the domain specific implementations are performing well.

Inherent in the domain specific methodology however is the limitation of the programmer's scope. By restricting the programmer to a particular application, the system implementer has a smaller subset of constructs to support, hence they can support a wider range of platforms. The domain specific abstraction hides this restriction by using a existing natural limitation of the domain programmer's scope, the application domain.

Hence, it is feasible to support portable, efficient implementations of a single task description.

9.1.2. Predictive Metric Models

To predict the run-time characteristics or metrics of the implementation of a task upon a platform, I considered what determines these run-time characteristics in Section 3.2. I observed that the task descriptions given by the programmer, which can be decomposed into domain functions and data types are fixed, and have a structure that can be known *a priori*. In Section 3.2.1, I noted that there was another category of inputs, implementation variables, which don't necessarily have meaning within the application domain, affect the metrics of the task, and can thus be varied by the system safely.

I answered the question of predicting domain metrics by constructing models based upon the nature of the domain task and the implementation variables in Section 3.2.2. In my case study in Chapter 6, I demonstrated the efficacy of these models for a range of platforms, for two domain metrics. I was able to create models that both incorporate additional information to become better predictors, but can also make predictions with a low bound on the error.

Hence, the run-time characteristics of tasks can be predicted for heterogeneous computing systems.

9.1.3. Partitioning of Workloads

The key conceptual issue relating to partitioning was how differing allocations of tasks and platforms could be interpreted in terms of the domain programmers' objectives, as reflected by the generalised mapping problem in Section 2.3.2. The solution that I propose in Section 3.3 is to make use of the domain metric abstraction, and to present different allocations of tasks to platforms as a trade-off between different metric values. Metric trade-offs are intuitive to the domain programmer, and hence enable the programmer to balance their objectives in response to the resources available.

To enable the representation of allocation as metric trade-offs, I formulated the allocation as multiple, multi-objective ILP programs, in Section 3.3.1, with each program representing a point in the Pareto design space. An important feature of these ILP programs, is that the metrics of multiple tasks and platforms are reconciled as defined within the domain. In Chapter 7, I demonstrated this approach in practice, generating a truthful metric representation of a large workload of derivative pricing tasks upon a broad set of heterogeneous platform.

Furthermore, these domain knowledge informed partitions achieve super-linear performance scaling. The performance observed is more than that predicted by the sum of ostensible performance of the different platforms. This is possible because the specialisation of the platforms is being considered as part of the partitioning.

Hence, programmers can partition their workloads so to balance their objectives.

9.2. Further Work

There are two promising areas for future research based upon this dissertation. Firstly, there is potential for the expansion of the case study, and secondly, there is the expansion of the broader methodology.

9.2.1. Expanding the case study

The breadth of the derivatives pricing case study can be expanded as part of future work. This expansion would mostly be of interest to domain programmers and financial engineers, extending the breadth of the domain supported.

Extending F^3

The utility of F^3 for computational finance would be enhanced by adding further underlying model and derivative product definitions. In particular, those underlying models which are computationally impractical except when using the Monte Carlo algorithm, such as the Hull-White-Merton model, would probably attract the most interest from the computational finance community.

A less modest refinement would be to consider a different pricing algorithm such as the Finite Difference Method. Adding the capability to the framework to automatically switch between using the Monte Carlo or Finite Difference Method where appropriate would empower financial engineers with the best of both algorithms.

Finally, a large expansion would be to consider backwards looking financial products. These are financial products which consider not just the underlying's price at expiry, but at all times throughout the product's lifetime. These are typically valued using the Least Squares Monte Carlo approach proposed by Longstaff and Schwartz [87]. This would be of interest from a computational point of view, as it would require multiple, memory-bound linear algebra operations, as opposed to the embarrassingly parallel compute-bound Monte Carlo considered in this project.

Additional Performance Metrics

In this dissertation, I chose to define the latency and accuracy metrics to be the makespan and 95% confidence interval respectively. There are multiple other definitions that could be used, or considered in parallel. In addition to considering the makespan, I could consider the latency distribution of the tasks, or individual deadlines for each task. For accuracy, in addition to alternative statistical properties of the price, there are the so-called "Greeks" measures that could be calculated for each option.

A further natural point of expansion of the metric models would be to experimentally verify the cost model outlined in Section 8.1.2. With the rising popularity of IaaS providers for HPC, pricing computations is an increasingly relevant consideration. An obstacle to taking advantage of this trend in this work is the limited variety of heterogeneous architectures available from such providers. In lieu of observable market prices, cost models can be used which incorporate both the capital cost of the technology used, as well as the energy used, such as the Simple model from the Uptime Institute [86].

Refined Partitioning Approaches

In this dissertation, I considered three broad approaches to partitioning: heuristics, global or numerical optimisation algorithms and Mixed Integer Linear Programming. However, to keep the evaluation to a reasonable length, I only considered two instances of the heuristic approach and one instance of the other two approaches, that I refer to as the domain approaches.

In terms of the heuristic approach, I considered heuristics that a domain programmer would most likely derive themselves. Hence, there is scope for considering more complex heuristics, such as those described in Braun et al's work [40]. I would start by considering the *min-min* heuristic, which Braun et al found to be the most broadly applicable.

Both the domain partitioning approaches have significant scope to be optimised. Both partitioners would benefit from a richer set of heuristics being used as a starting allocation in both instances. Furthermore, both partitioners can be "tuned", although this would need to be done with some care as so as to avoid overfitting to a particular set of tasks and platforms. Finally, both partitioners would no doubt benefit from being executed in parallel, as opposed to the sequential implementations that I used.

9.2.2. Growing the Domain Specific Methodology

Beyond the derivatives pricing case study, more broadly, there is scope to enhance the domain specific methodology beyond the derivatives pricing instance of it that I have described in this dissertation.

More Case Studies

This work would be enhanced by further case studies, from other domains, particularly those orthogonal to the computational finance domain that I have considered. Doing so would serve two purposes: it would firstly rebut the criticism that the derivatives pricing domain is a particular outlier, and secondly, it could provide data to verify the claim that a small group of functions are used disproportionately frequently within a domain.

A more systematic approach would be to consider the programming motifs identified by Asanovic et al [65], and find a domain or multiple domains that contain functions that have algorithmic implementations that cover all of the motifs. In particular, I believe that subdomains with the broader area of Linear Algebra would yield a large degree of algorithmic variation. Doing so would provide considerable insight into the conditions under which a domain specific approach delivers portable efficiency, predictability and partitionablity.

Platform Specific Metric Models

A further enhancement of the methodology related to the domain metric models would be to consider models that are specific to a particular platform. In my case study, I found that linear or relatively simple polynomial models could be used effectively for multiple platforms. However, I would expect that in the case of memory or communication bound algorithms, platform specific models would offer greater predictive power, given the radical differences between architectures.

Partitioning + Scheduling = Mapping

As outlined in Section 8.2.3, the partitioning feature that has been proven in this dissertation can be broadened into the more general mapping feature described by Braun et al [38] by supporting scheduling.

The chief challenge in implementing mapping would be the latencies of tasks that would be required to do so. This requires estimates of task latencies, which might not be feasible in all cases. A further consideration would be the impact upon the computational intensity of the now mapping problem, as the results in Section 7.2.1 demonstrate for the case of partitioning.

9.3. Reflection

Finally, I conclude this dissertation by considering the broader implications of the results reported. Throughout this research, I have regularly reflected on how abstraction must be one of the most unintuitive concepts in computing.

On face value, abstraction is an arbitrary set of rules that are created that the programmer must master so they can then express their intentions as programs. This is not for the benefit of the computing system however, as these rules often require a significant amount of infrastructure so that programs written using the rules can be executed upon the computing platform.

However, the benefit provided by an abstraction is very powerful. It provides a set of shared assumptions that both the programmer and system designer can build upon. This strategy for hiding complexity has been amazingly successful, enabling millions to access Von Neumann-based computer systems with limited understanding of how these platforms function.

I believe that the results of this dissertation suggest that this strategy can be extended to heterogeneous computing, however there will have to compromises made by both system developers and programmers. I believe that concepts such as domain specificity can ease these compromises, and are worthy of further attention.

Bibliography

[1] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," in *Proceedings of the European Conference on Object Orientated Programming (ECOOP 2008)*, pp. 76–103, Springer Berlin Heidelberg, 2008.

[2] M. Fowler, *Domain-specific languages.* Pearson Education, 2010.

[3] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, pp. 316–344, 2005.

[4] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A Domain-Specific Approach to Heterogeneous Parallelism," in *Proceedings of Principles and Practices of Parallel Programming (PPOPP 2011)*, pp. 35–46, 2011.

[5] "Stratix V Device Overview," 2015.

[6] Xilinx Corporation, *Virtex-6 Family Overview*, 2012.

[7] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," 2005.

[8] R. Hartenstein, "The von Neumann Syndrome," in *Proceedings of the Stamatis Vassiliadis Memorial Symposium*, pp. 1–7, 2010.

[9] S. Moore, "Multicore is bad news for supercomputers," *IEEE Spectrum*, vol. 45, 2008.

[10] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of American Federation of Information Processing Societies (AFIPS) Spring Joint Computing Conference (AFIPS 1967)*, (New York, New York, USA), pp. 483–485, ACM Press, Apr. 1967.

[11] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: optimizing MapReduce on heterogeneous clusters," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, (New York, NY, USA), pp. 61–74, ACM, 2012.

[12] NVIDIA Inc., *CUDA C Programming Guide*, 2015.

[13] Xilinx Corporation, *Vivado Design Suite User Guide*, 2014.

[14] J. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science Engineering*, vol. 12, pp. 66 –73, May-June 2010.

[15] "OpenSPL: Revealing the Power of Spatial Computing," tech. rep., The OpenSPL Consortium, 2013.

[16] "The OpenACC Application Programming Interface," tech. rep., The OpenACC Consortium, 2013.

[17] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO 2009)*, pp. 45–55, 2009.

[18] W. Luk, J. Coutinho, T. Todman, Y. Lam, W. Osborne, K. Susanto, Q. Liu, and W. Wong, "A High-level Compilation Toolchain for Heterogeneous Systems," in *Proceedings of IEEE System on Chip Conference (SOCC 2009)*, pp. 9 –18, 2009.

[19] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA 2011)*, (New York, NY, USA), pp. 33–36, ACM, 2011.

[20] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, "EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System," *SIGPLAN Notes*, vol. 42, pp. 156–166, June 2007.

[21] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI 2004)*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[22] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a high-level language for GPUs," in *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2012)*, vol. 47, (New York, New York, USA), p. 1, ACM Press, June 2012.

[23] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Notes*, vol. 35, pp. 26–36, June 2000.

[24] S. Kelly and R. Pohjonen, "Worst Practices for Domain-Specific Modeling," *IEEE Software*, vol. 26, pp. 22–29, July 2009.

[25] J. L. C. Izquierdo and J. Cabot, "Community-driven language development," in *International Workshop on Modeling in Software Engineering (MISE 2012)*, pp. 29–35, IEEE, June 2012.

[26] T. Kosar, P. E. M. Lopez, P. A. Barrientos, and M. Mernik, "A Preliminary Study on Various Implementation of Domain-specific Language," *Information and Software Technology*, vol. 50, no. 5, pp. 390 – 405, 2008.

[27] A. F. Blackwell and C. Morrison, “A logical mind, not a programming mind: Psychology of a professional end-user,” in *Proceedings of the Workshop of Psychology of Programming Interest Group (PPIG 2010)*, pp. 19–21, 2010.

[28] B. A. Nardi, *A Small Matter of Programming: Perspectives on End User Computing.* MIT Press, 1993.

[29] A. J. Ko, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, S. Wiedenbeck, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, and H. Lieberman, “The state of the art in end-user software engineering,” *ACM Computing Surveys*, vol. 43, pp. 1–44, Apr. 2011.

[30] R. E. Johnson and B. Foote, “Designing reusable classes,” *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, June-July 1988.

[31] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users’ Guide.* Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[32] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, “Basic linear algebra subprograms for fortran usage,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[33] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. K. Prasanna, M. Püschel, and M. Veloso, “SPIRAL: Automatic implementation of signal processing algorithms,” in *Proceedings of High Performance Embedded Computing (HPEC 2000)*, 2000.

[34] D. B. Thomas and W. Luk, “A Domain Specific Language for Reconfigurable Path-based Monte Carlo Simulations,” in *Proceedings of International Conference on Field-Programmable Technology (ICFPT 2007)*, pp. 97–104, 2007.

[35] J. Cong, G. Reinman, A. Bui, and V. Sarkar, “Customizable Domain-Specific Computing,” *IEEE Design and Test of Computers*, vol. 28, pp. 6 –15, March-April 2011.

[36] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, “Green-Marl: a DSL for easy and efficient graph analysis,” *SIGARCH Computer Architecture News*, vol. 40, pp. 349–362, Mar. 2012.

[37] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun, “Language Virtualization for Heterogeneous Parallel Computing,” *SIGPLAN Notes*, vol. 45, pp. 835–847, 2010.

[38] T. Braun, H. Siegel, and A. Maciejewski, “Heterogeneous Computing: Goals, Methods, and Open Problems,” in *High Performance Computing* (B. Monien, V. Prasanna, and S. Vajapeyam, eds.), vol. 2228 of *Lecture Notes in Computer Science*, pp. 307–318, Springer Berlin/Heidelberg, 2001.

[39] A. Khokhar, V. Prasanna, M. Shaaban, and C.-L. Wang, “Heterogeneous computing: challenges and opportunities,” *Computer*, vol. 26, pp. 18–27, June 1993.

[40] T. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Muthucumaru, A. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 61, pp. 810–837, June 2001.

[41] N. Fisher, J. Anderson, and S. Baruah, "Task Partitioning upon Memory-Constrained Multiprocessors," in *Proceedings of the IEEE International Conference on Embedded and Real-Time Computer Systems and Applications (RTCSA 2005)*, pp. 416–421, IEEE, 2005.

[42] W. W. Chu, L. J. Holloway, M.-t. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *Computer*, 1980.

[43] F. Berman and R. Wolski, "Application-level scheduling on distributed heterogeneous networks," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC 1996)*, IEEE, 1996.

[44] D. Grewe and M. F. P. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in *Proceedings of the international conference on Compiler construction (CC/ETAPS 2011)*, vol. 6601, pp. 286–305, 2011.

[45] Q. Kang, H. He, and H. Song, "Task assignment in heterogeneous computing systems using an effective iterated greedy algorithm," *Journal of System Software*, vol. 84, pp. 985–992, June 2011.

[46] O. H. Ibarra and C. E. Kim, "Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors," *Journal of the ACM*, vol. 24, pp. 280–289, Apr. 1977.

[47] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization (IISWC 2009)*, pp. 44–54, IEEE, Oct. 2009.

[48] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, pp. 1–25, October 2014.

[49] K. L. Spafford and J. S. Vetter, "Aspen: a domain specific language for performance modeling," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2012)*, p. 84, IEEE, Nov. 2012.

[50] D. Ardagna, E. D. Nitto, P. Milano, D. Petcu, C. Sheridan, C. Ballagny, F. D. Andria, and P. Matthews, "MODACLOUDS : A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds," in *Proceedings of the International Workshop on Modeling in Software Engineering (IWMSE 2012)*, pp. 50–56, IEEE, June 2012.

[51] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson, "Estimating Service Resource Consumption From Response Time Measurements," in *Proceedings on International ICST Conference on Performance Evaluation Methodologies and Tools*, p. 48, ICST, Oct. 2009.

[52] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[53] K. M. Tarplee, R. Friese, A. A. Maciejewski, and H. J. Siegel, "Scalable linear programming based resource allocation for makespan minimization in heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 84, pp. 76–86, Oct. 2015.

[54] R. M. Karp, "Reducibility among combinatorial problems," *50 Years Integer Programing, 1958-2008: From Early Years to State-of-the-Art*, pp. 219–241, 2010.

[55] K. Sajjapongse, X. Wang, and M. Becchi, "A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with GPUs," *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC 2013)*, pp. 179–190, June 2013.

[56] O. Beaumont and L. Marchal, "Analysis of dynamic scheduling strategies for matrix multiplication on heterogeneous platforms," in *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing (HPDC 2014)*, pp. 141–152, ACM, 2014.

[57] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[58] Z. Wang, D. Grewe, and M. F. P. O'boyle, "Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, pp. 1–26, Dec. 2014.

[59] Shiann-Rong Kuang, Chin-Yang Chen, and Ren-Zheng Liao, "Partitioning and Pipelined Scheduling of Embedded System Using Integer Linear Programming," in *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS 2005)*, vol. 2, pp. 37–41, IEEE, 2005.

[60] G. R. Mudalige, M. B. Giles, C. Bertolli, and P. H. Kelly, "Predictive modeling and analysis of OP2 on distributed memory GPU clusters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, p. 61, 2012.

[61] N. Baltas and T. Field, "Software Performance Prediction with a Time Scaling Scheduling Profiler," in *Proceedings of the IEEE Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 107–116, IEEE, 2011.

[62] R. E. Bixby, "Solving Real-World Linear Programs: A Decade and More of Progress," *Operations Research*, vol. 50, pp. 3–15, Feb. 2002.

[63] R. Bixby and E. Rothberg, "Progress in computational mixed integer programming—A look back from the other side of the tipping point," *Annals of Operations Research*, vol. 149, pp. 37–41, Jan. 2007.

[64] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, pp. 519–539, ACM, June 2013.

[65] K. Asanovic, B. C. Catanzaro, D. A. Patterson, and K. A. Yelick, "The Landscape of Parallel Computing Research : A View from Berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department University of California Berkeley, 2006.

[66] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, Sept. 1972.

[67] OpenMP Architecture Review Board, "OpenMP," 1998-.

[68] B. Lewis and D. J. Berg, *Multithreaded Programming with Pthreads.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[69] J. Reinders, *Intel Threading Building Blocks.* Sebastopol, CA, USA: O'Reilly & Associates, Inc., first ed., 2007.

[70] Z. Whang and M. F. O'Boyle, "Partitiong streaming parallelism for multi-cores: a machine learng based approach," in *Proceedings of International Conference on Parallel Architecture and Compilation Techniques (PACT 2010)*, pp. 307–318, Vienna: ACM, 2010.

[71] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*, vol. 44, (New York, New York, USA), p. 177, ACM Press, June 2009.

[72] G. Kirlik and S. Sayın, "A new algorithm for generating all nondominated solutions of multiobjective discrete optimization problems," *European Journal of Operations Research*, vol. 232, no. 3, pp. 479–488, 2014.

[73] S. Mai, "The global derivatives market - an introduction," white paper, Deutsche Börse AG, 60485 Frankfurt /Main Germany, April 2008.

[74] J. C. Hull, *Options, Futures and Other Derivatives.* Pearson, 8th edition ed., 2011.

[75] B. McLean and J. Nocera, *All The Devils Are Here: Unmasking the Men Who Bankrupted the World.* Penguin Books Limited, 2011.

[76] S. L. Heston, "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options," *The Review of Financial Studies*, vol. 6, no. 2, pp. pp. 327–343, 1993.

[77] S. Tazukwa and P. L'Ecuyer, "Efficient and Portable Combined Tausworth Random Number Generators," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, pp. 99–112, Apr. 1991.

[78] G. Box and M. E. Muller, "A Note on the Generation of Random Normal Deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 1958.

[79] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings - Computer and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[80] A. H. Tse, D. B. Thomas, K. H. Tsoi, and W. Luk, "Efficient reconfigurable design for pricing asian options," *SIGARCH Computer Architecture News News*, vol. 38, pp. 14–20, Jan. 2011.

[81] C. de Schryver, I. Shcherbakov, F. Kienle, N. Wehn, H. Marxen, A. Kostiuk, and R. Korn, "An Energy Efficient FPGA Accelerator for Monte Carlo Option Pricing with the Heston Model," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig 2011)*, pp. 468 –474, 30 2011-dec. 2 2011.

[82] Intel Corporation, "The Intel Xeon Phi Product Family," 2013.

[83] V. W. Lee, P. Hammarlund, R. Singhal, P. Dubey, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, and S. Chennupaty, "Debunking the 100X GPU vs. CPU myth," *ACM SIGARCH Computer Architecture News*, vol. 38, p. 451, June 2010.

[84] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001–.

[85] L. Perron and M. A. Trick, eds., *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, vol. 5015 of *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[86] J. Koomey, K. Brill, P. Turner, J. Stanley, and B. Taylor, "A Simple Model for Determining True Total Cost of Ownership for Data Centers," tech. rep., 2008.

[87] F. A. Longstaff and E. S. Schwartz, "Valuing american options by simulation: a simple least-squares approach," *Review of Financial studies*, vol. 14, no. 1, pp. 113–147, 2001.

Appendices

A. Full Experimental Procedures and Results

A.1. Experimental Platform Details

A.1.1. CPU Platforms

Tables A.1 and A.2 provide the details of all of the CPU Experimental platforms used throughout Part II.

Table A.1.: Experimental CPU and GPU Platform Resources

Vendor	Name	Programming Standard (Tool)
Intel	Core i7-2600	POSIX (GCC 4.8.2)
Intel	Xeon E5-2680v1	POSIX (GCC 4.8.2)
Intel	Xeon E5-2680v2	POSIX (GCC 4.8.2)
Intel	Xeon E5-2670v2	POSIX (GCC 4.8.2)
AMD	Opteron 6272	POSIX (GCC 4.8.2)
ARM	11 76JZF-S	POSIX (GCC 4.6.3)

Table A.2.: Experimental CPU Platform Resources

Name	Release Date (mm/yy)	CMOS size (nm)	Clock Rate (GHz)	Cores	SIMD Width	L1 Cache Size (kB)	Theoretical Peak Performance (GFLOPs)
Core i7-2600S	09/2011	45	2.8	4	4	32	44.80
Xeon E5-2680v1	01/2012	22	2.70	8	4	32	86.40
Xeon E5-2680v2	07/2013	22	2.80	8	4	32	89.60
Xeon E5-2670v2	07/2013	22	2.80	4	4	32	44.80
Opteron 6272	11/2011	32	2.10	8	4	32	268.80
11 76JZF-S	07/2004	45	0.70	1	1	32	0.70

A.1.2. GPU Platforms

Tables A.3 and A.4 provide the details of all of the GPU Experimental platforms used throughout Part II.

Table A.3.: Experimental GPU Platform Resources

Vendor	Name	Programming Standard (Tool)
NVIDIA	Quadro K4000	OpenCL (NVIDIA OpenCL SDK 6.5)
NVIDIA	GK104	OpenCL (NVIDIA OpenCL SDK 6.5)
AMD	Firepro W5000	OpenCL (AMDAPP 2.9)
Intel	Xeon Phi 3120P	OpenCL (Intel SDK for OpenCL 4.4)

Table A.4.: Experimental GPU Platform Resources

Name	Release Date (mm/yy)	CMOS size (nm)	Clock Rate (GHz)	OpenCL Compute Units	OpenCL Processing Elements	Local Memory Size (kB)	Theoretical Peak Performance (GFLOPs)
Quadro K4000	03/2013	28	0.80	4	768	48	1244.16
GK104	03/2013	28	0.80	8	1538	48	2448.38
Firepro W5000	06/2012	28	0.83	12	768	32	1267.20
Xeon Phi 3120P	06/2013	22	1.10	224	224	32	2002.75

A.1.3. FPGA Platforms

Tables A.5 and A.6 provide the details of FPGA Experimental platforms used throughout Part II.

Table A.5.: FPGA Experimental Platforms

Vendor	Name	FPGA	Communication Technology	Programming Standard (Tool)
Xilinx	ZC706 1.1	Xilinx Zynq 7Z045	AXI	Xilinx C (Xilinx Vivado HLS 2013.4)
Nallatech	P385-A7	Altera Stratix V GXA7	PCIe	OpenCL (Altera OpenCL SDK 13.0)
Nallatech	P385-D5	Altera Stratix V GSD5	PCIe	OpenCL (Altera OpenCL SDK 14.0)
Maxeler	Max 3424A	Xilinx Virtex 6 475T	PCIe	OpenSPL (Maxeler MaxCompiler 14.1)
Maxeler	Max 4	Altera Stratix V GSD8	PCIe	OpenSPL (Maxeler MaxCompiler 14.1)

Table A.6.: Experimental FPGA Resources [5, 6]

FPGA	CMOS Size (nm)	Release Date	Targeted Clockrate (MHz)	LookUp Tables (LUTs)	Flipflop Registers (FFs)	Block RAMs (BRAMs)	DSPs
Zynq 7Z045	28	03/2012	100	218.6k	437.2k	545	900
Stratix V GXA7	28	03/2012	250	622k	939k	2304	768
Stratix V GSD5	28	03/2012	250	457k	690k	2014	3180
Virtex 6 XC6VSX475T	40	02/2009	200	297.6k	595.2k	1064	2016
Stratix V GSD8	28	03/2012	180	695k	1050k	2567	3926

A.1.4. Network Locations

Table A.7 provides an overview of the network locations used in the experimental platforms in Chapters 6 and 7.

Table A.7.: Network Location Characteristics

Network Name	Network Location	Geographic Location	Network RTT (mS)
Localhost	Localhost	ICL, London, UK	0.024
ICL EEE Workshop	LAN	ICL, London, UK	0.268
ICL Datacentre	LAN	ICL, London, UK	0.380
ICL Insecure subnet	LAN	ICL, London, UK	2.463
AWS East	WAN	AWS, USA East Region	88.538
AWS West	WAN	AWS, USA West Region	158.339
GCE Central	WAN	GCE, USA Central Region	111.232
UCT Datacentre	WAN	UCT, Cape Town, ZA	3300.000

A.2. Tasks

Table A.8 provides the details of the option pricing tasks used throughout Part II.

The computational operation values were calculated using instruction counting in a sequential, OpenCL version of the F^3 code. The values assuming that 4096 time steps were considered per simulation path.

Table A.8.: Overview of computational intensity of option pricing tasks

Option Task Name	Underlying	Option	Computational Operations $\left(\frac{\text{kFLOP}}{\text{path}}\right)$
BS-A	Black Scholes	Asian	139.267
BS-B		Barrier	139.266
BS-DB		Double Barrier	143.360
BS-DDB		Digital Double Barrier	143.361
H-A	Heston	Asian	319.492
H-B		Barrier	319.491
H-DB		Double Barrier	323.585
H-DDB		Digital Double Barrier	323.586
H-E		European	315.395

A.3. Measurement

A.3.1. Latency

In all instances where latency (L) is measured wall clock time is used. This is implemented as a check of the system's time T_{start} at the start of the computation, and a check at the end of the target T_{end}, with the latency being the difference the two measurements, as given in (A.1).

$$L = T_{start} - T_{end} \tag{A.1}$$

When the latency of more than one task is being considered, the latency reported will be of that of the longest running task, i.e. the makespan of the task workload.

A.3.2. Accuracy

When accuracy is quoted, it is referring to the 95% confidence interval of the option pricing result, as given in (A.2).

$$A_{95\%} = 1.96\frac{\sigma}{\sqrt{n}} \tag{A.2}$$

Similar to latency, when the accuracy of more than one task is being considered, the largest value thereof is reported, i.e. the least accurate task.

A.4. Portable Execution

A.4.1. Parallel Scaling

The results plotted in Section 5.2.2 are reported in Table A.9.

Table A.9.: Parallel scaling experiment performance results

Type	Designation	Baseline Latency (s)	Parallel Latency (s)	Parallel Scaling Factor
CPUs	Desktop	3366.30	710.55	4.74
	Server	2763.16	263.26	10.50
	Manycore	4941.23	124.35	39.74
GPUs	NVIDIA Workstation	44.91	12.63	3.56
	NVIDIA Cloud	45.08	6.73	6.70
	AMD Workstation	162.06	14.62	11.08
	Phi	2685.96	40.17	66.87
FPGAs	SoC	1144.28	906.77	1.26
	PCIe-A7	281.24	31.53	8.92
	PCIe-D5	190.44	19.45	9.79
	Max3	262.07	27.99	9.36
	Max4	235.94	20.39	11.57

A.4.2. Absolute Performance

The results plotted in Section 5.2.3 are reported in Tables A.10, A.11 and A.12.

Table A.10.: Absolute performance results for multicore CPUs

	Option Designation	Sequential POSIX	Desktop	Server	Manycore	External [80, 81]
Throughput (GFLOPs)	KSO1	0.63	2.59	6.91	16.04	4.90
	KSO2	1.48	6.26	15.99	34.85	4.96
	KSO3	1.13	4.74	12.32	27.93	4.96
	KSO4	1.02	1.49	12.77	26.87	5.03
	KSO5	0.74	3.35	8.91	19.39	5.03
	KSO6	1.11	4.42	12.30	26.60	5.03
	KSO7	2.03	8.98	22.45	48.57	5.03
	KSO8	1.25	5.81	14.76	29.98	5.03
	KSO9	0.73	3.33	8.93	19.56	5.03
	KSO10	1.26	5.73	14.97	32.65	5.03
	KSO11	0.62	2.10	6.92	15.23	4.96
	KSO12	1.06	4.61	12.16	27.11	5.03
	IC	0.74	3.02	7.42	16.12	2.96

Table A.11.: Absolute performance results for GPUs

	Option Designation	Sequential OpenCL	NVIDIA Workstation	NVIDIA Cloud	AMD Workstation	Phi	External [80, 81]
Throughput (GFLOPs)	KSO1	7.99	302.32	589.88	450.19	1311.47	27.32
	KSO2	3.64	288.58	565.10	308.92	452.93	27.66
	KSO3	3.95	288.82	564.51	307.50	451.83	27.66
	KSO4	3.90	296.26	579.69	290.49	454.98	28.01
	KSO5	4.43	296.18	580.14	290.99	615.82	28.01
	KSO6	4.00	295.32	579.23	290.25	464.33	28.01
	KSO7	3.52	296.57	579.36	290.91	402.72	28.01
	KSO8	3.76	296.63	580.43	289.66	441.31	28.01
	KSO9	4.46	297.14	580.75	290.04	692.34	28.01
	KSO10	3.82	295.61	579.07	290.56	439.76	28.01
	KSO11	5.13	289.20	565.46	307.97	1033.49	28.01
	KSO12	3.97	298.79	584.23	293.11	471.65	28.01
	IC	4.49	173.41	337.77	281.86	832.45	29.66

Table A.12.: Absolute performance results for FPGAs

	Option Designation	Sequential OpenCL	SoC	P385-A7	P385-D5	Max3	Max4	External [80, 81]
Throughput (GFLOPs)	KSO1	7.99	4.29	99.59	263.21	111.95	165.16	11.28
	KSO2	3.64	3.42	110.33	228.88	113.32	166.99	11.42
	KSO3	3.95	3.42	110.33	228.27	112.25	168.58	11.42
	KSO4	3.90	3.46	101.56	146.92	113.33	168.96	11.56
	KSO5	4.43	3.46	101.56	146.54	114.01	170.76	11.56
	KSO6	4.00	3.46	101.56	147.15	114.35	171.07	11.56
	KSO7	3.52	3.46	101.56	146.82	113.90	170.86	11.56
	KSO8	3.76	3.46	101.56	146.85	114.33	170.72	11.56
	KSO9	4.46	3.46	101.56	147.16	114.47	170.01	11.56
	KSO10	3.82	3.46	101.56	147.03	113.88	170.42	11.56
	KSO11	5.13	3.42	110.33	228.40	112.61	166.66	11.42
	KSO12	3.97	3.46	108.76	157.02	113.89	168.89	11.56
	IC	4.49	2.51	49.99	207.70	111.25	160.72	71.42

A.5. Predictive Model Evaluation

A.5.1. Incorporation

The results reported in Sections 6.2.2 and 6.2.3 are reported in full in Tables A.13 and A.14.

Table A.13.: Latency model incorporation results.

Benchmark Paths / Run-time Paths	0.03125	0.0625	0.125	0.25	0.5	1.0
Designation	Mean Relative Error (%)					
Desktop	20.74	14.63	10.73	9.78	5.90	1.11
Local Server	1938.18	162.29	72.40	12.49	6.09	1.56
Local Pi	3.04	1.13	0.53	0.30	0.21	0.04
AWS Server EC1	532.84	142.36	26.74	12.39	5.93	1.13
AWS Server EC2	77.74	11.35	4.14	1.91	0.92	0.17
GCE Server	548.80	25.13	8.43	3.59	1.25	0.24
AWS Server WC1	216.22	23.21	5.03	2.63	1.22	0.23
AWS Server WC2	83.89	8.76	3.62	1.65	0.69	0.13
Remote Server	221.32	69.76	25.96	9.08	2.67	0.50
CPUs	130.74	22.60	8.27	3.68	1.70	0.33
Local GPU 1	3973.24	360.13	118.33	28.19	10.03	1.27
Local GPU 2	3654.98	299.95	29.65	15.92	8.24	2.65
AWS GPU EC	603.08	119.08	16.30	8.53	3.46	0.71
AWS GPU WC	329.47	85.79	11.29	6.95	2.45	0.46
Remote Phi	314.81	131.45	48.80	39.43	35.55	5.55
GPUs	980.97	170.73	31.60	16.00	7.57	1.44
Local FPGA 1	5109.49	1452.11	451.64	377.04	106.75	28.95
Local FPGA 2	2431.83	559.50	151.68	64.26	39.51	4.87
FPGAs	3524.97	901.36	261.73	155.65	64.94	11.87

Table A.14.: Accuracy model incorporation results.

Benchmark Paths / Run-time Paths		0.125			0.25			0.5			1.0	
Designation		Min Relative Error(%)					Mean Relative Error(%)				Max Relative Error(%)	
BS-A	28.36	146.24	203.72	8.62	74.57	114.06	23.88	37.08	51.25	0.54	4.16	53.45
BS-B	17.70	141.34	182.43	15.80	80.74	99.85	36.64	39.24	41.34	0.06	1.56	57.37
BS-DDB	105.07	207.89	791.25	47.43	112.32	530.21	4.70	36.15	345.62	0.74	11.86	215.10
BS-DB	150.99	172.24	201.62	78.70	92.89	113.15	26.51	36.16	50.66	0.71	3.46	10.51
H-A	121.61	168.84	199.54	58.15	90.33	111.76	12.11	33.99	49.68	0.12	3.26	20.66
H-B	18.76	115.74	213.84	10.07	61.63	121.35	1.72	20.64	62.67	0.74	9.96	73.39
H-DB	2.00	107.73	327.56	17.40	62.60	194.39	0.92	19.50	107.26	0.77	14.74	76.34
H-DDB	176.14	184.60	193.09	95.34	101.17	107.33	38.13	42.17	46.42	0.05	0.72	3.52
H-E	167.10	174.99	184.58	89.27	94.49	100.57	33.90	37.54	41.82	0.28	1.94	5.30

A.5.2. Extrapolation

The results reported in Sections 6.2.2 and 6.2.3 are reported in full in Tables A.15 and A.16.

Table A.15.: Latency model extrapolation results. The ordering is the minimum, geometric mean and maximum of the results.

Run-time Paths / Benchmark Paths	1.0	2.0	4.0	8.0	16.0
Designation	Mean Relative Error (%)				
Desktop	0.70	2.40	2.51	3.47	3.81
Local Server	0.39	1.13	1.35	1.27	1.30
Local Pi	0.02	1.45	1.81	1.91	1.94
AWS Server EC1	0.49	1.23	1.47	1.54	1.56
AWS Server EC2	0.15	0.46	0.50	0.52	0.70
GCE Server	0.26	0.77	0.87	0.84	0.90
AWS Server WC1	0.16	0.51	0.61	0.79	0.74
AWS Server WC2	0.14	0.47	0.67	0.68	0.75
Remote Server	0.47	7.01	14.19	23.86	39.38
CPUs	0.22	1.12	1.42	1.61	1.81
Local GPU 1	0.67	1.08	0.99	0.87	0.93
Local GPU 2	0.60	2.27	1.96	1.94	1.96
AWS GPU EC	0.18	0.46	0.73	0.84	0.92
AWS GPU WC	0.15	0.62	0.75	0.89	1.04
Remote Phi	0.87	21.87	39.76	53.42	66.16
GPUs	0.39	1.73	2.12	2.32	2.58
Local FPGA 1	2.62	6.54	8.33	8.60	8.74
Local FPGA 2	1.34	5.40	5.85	6.54	7.10
FPGAs	1.87	5.94	6.99	7.50	7.88

Table A.16.: Accuracy model extrapolation results.

Run-time Paths / Benchmark Paths	1.0			2.0			4.0			8.0			16.0		
Designation		Min Relative Error(%)					Mean Relative Error(%)						Max Relative Error(%)		
BS-A	0.58	4.02	37.31	0.57	4.67	27.27	0.92	4.81	22.43	1.02	4.59	28.64	0.91	3.88	34.27
BS-B	0.05	1.58	43.88	0.12	1.11	3.20	0.73	2.07	17.47	0.53	2.28	39.60	0.31	2.21	34.05
BS-DB	0.24	2.27	15.80	0.26	1.66	18.80	0.42	2.70	17.91	0.90	2.91	18.29	0.91	2.58	15.61
BS-DDB	0.15	2.73	14.61	0.65	2.96	14.29	1.02	3.00	13.53	1.15	4.22	12.46	1.52	3.68	12.64
H-A	0.24	4.04	14.89	0.47	3.49	17.76	0.46	3.20	19.04	0.19	3.65	27.50	0.35	2.56	10.83
H-B	0.15	6.76	63.50	0.14	6.81	58.92	0.10	6.05	59.20	0.10	5.42	63.96	0.22	5.22	65.79
H-DB	0.78	11.79	80.61	0.96	9.34	86.30	0.93	7.96	73.09	0.05	6.04	71.35	0.16	7.00	71.07
H-DDB	0.16	2.25	17.65	0.03	0.64	12.40	0.22	1.08	13.90	0.32	1.46	17.12	0.10	1.11	14.82
H-E	0.79	3.10	8.48	1.02	2.35	3.83	0.79	2.03	3.45	1.07	1.86	3.88	1.04	2.25	3.53

A.6. Partitioner Characterisation

A.6.1. Partitioner Implementations

Heuristic Partitioners

Listing A.1 gives the implementation of the Best Platform heuristic, as implemented in F^3.

Listing A.1: Implementation of best platform allocation heuristic

```
def best_platform_allocation(delta,gamma):
    """
    delta,gamma are numpy matrices of shape mu times tau
    return value allocation is matrix also of shape mu times tau
    """
    best_platform_latencies = numpy.sum(delta + gamma, axis=1)
    best_platform = numpy.argmin(best_platform_latencies)
    allocation = numpy.zeros(delta.shape)
    allocation[best_platform,:] = 1.0

    return allocation
```

Listing A.2 gives the implementation of the proportional allocation heuristic, as implemented in F^3.

Listing A.2: Implementation of proportional allocation heuristic

```
def proportional_allocation(delta,gamma):
    """
    delta,gamma are numpy matrices of shape mu times tau
    return value allocation is matrix also of shape mu times tau
    """
    platform_latencies = numpy.sum(delta + gamma, axis=1)
    platform_proportions = numpy.sum(platform_latencies**0.5)
                /platform_latencies
    allocation = numpy.ones(delta.shape)*platform_proportions

    return allocation
```

MILP Partitioner

Listing A.3 gives the ZIMPL code as used within $F^3$to describe partitioning problems to SCIP[85].

Listing A.3: ZIMPL implementation of allocation problem

```
set PLATFORMS := {read PLATFORM_FILE as "<1s>"};
set TASKS := {read TASK_FILE as "<1s>"};
set PT:= PLATFORMS * TASKS;

#Problem parameters
param DELTA[PT] := read LATENCY_PP_FILE as "<1s,2s> 3n";
param GAMMA[PT] := read LATENCY_SETUP_FILE as "<1s,2s> 3n";

#Problem Variables
var A[PT] real >= 0;  # Allocation proportional matrix
var GL real; # Maximum platform latency
var B[PT] binary; # Allocation binary matrix

#Objective Function
minimize latency: GL;

#Task completion constraint
subto task_complete:
  forall <t> in TASKS:
    sum <p> in PLATFORMS: A[p,t] >= 1.0;

#Constraint thresholding the allocation binary matrix
subto platform_use:
  forall <p,t> in PT:
    B[p,t] >= A[p,t];

#Constraint for implementing the max function
subto con_task_max:
  forall <p> in PLATFORMS:
    (sum <t> in TASKS:
      (A[p,t] * DELTA[p,t] + B[p,t] * GAMMA[p,t])) <= GL;
```

A.6.2. Synthetic Characterisation Results

The results plotted in Section 7.2.1 are given in full in Tables A.17, A.18,A.19 and A.20.

Latency

Table A.17.: Synthetic domain partitioner time characterisation with respect to problem size. ML is the machine learning-based partitioner, while MILP is the Mixed Integer Linear Programming Partitioner.

Case	1		2		3		4	
Partitioner	ML	MILP	ML	MILP	ML	MILP	ML	MILP
Variables								
4	360.71	0.07	360.68	0.03	240.49	0.04	180.40	0.05
8	449.94	0.03	503.73	0.04	477.18	0.04	480.89	0.04
16	434.79	0.06	460.68	0.06	533.74	0.05	507.06	0.06
32	381.73	0.13	402.29	0.10	433.71	0.09	113.42	0.10
64	118.44	0.42	99.56	0.23	231.53	0.26	220.66	0.26
128	31.57	1.11	95.71	0.70	202.49	0.73	258.26	0.58
256	45.37	4.78	58.88	2.34	140.38	1.97	468.33	1.55
512	31.87	62.56	91.83	48.47	128.86	10.18	587.10	4.05
1024	52.73	422.76	81.95	522.42	105.47	235.87	604.56	46.32
2048	152.86	611.58	150.00	610.74	201.62	611.09	608.55	576.32
4096	449.81	607.85	618.75	608.37	622.81	608.62	628.06	609.75

Table A.18.: Synthetic domain partitioner time characterisation with respect to problem linearity. NO is the numerical optimiser-based partitioner, while MILP is the Mixed Integer Linear Programming Partitioner.

Case	1		2		3		4	
Partitioner	ML	MILP	ML	MILP	ML	MILP	ML	MILP
ψ								
10^{-5}	601.09	0.24	601.10	0.26	601.07	0.22	-	-
10^{-4}	601.06	0.22	601.12	0.28	601.07	0.22	601.10	0.21
10^{-3}	601.11	0.46	601.12	0.32	601.10	0.30	601.11	0.32
10^{-2}	601.12	1.24	601.14	0.95	601.10	0.62	601.12	0.46
10^{-1}	601.09	27.74	535.16	5.22	601.11	1.96	601.14	1.54
1	1.80	28.27	62.81	18.36	2.50	4.73	541.27	3.01
10^{1}	62.32	2.68	305.00	1.24	63.16	0.75	2.60	0.37
10^{2}	61.90	1.82	365.15	0.75	183.62	0.39	2.59	0.23
10^{3}	121.93	2.10	365.28	0.78	183.58	0.32	2.27	0.23
10^{4}	122.40	1.54	365.24	1.19	183.08	0.57	2.25	0.23
10^{5}	122.87	1.43	365.48	1.77	183.67	2.86	2.47	1.08

Quality

Table A.19.: Synthetic domain partitioner quality characterisation with respect to problem size. NO is the numerical optimiser-based partitioner, while MILP is the Mixed Integer Linear Programming Partitioner.

Case	1		2		3		4	
Partitioner	ML	MILP	ML	MILP	ML	MILP	ML	MILP
Variables								
4	1.00	1.74	1.00	1.94	1.00	1.22	1.00	1.54
8	1.07	2.20	1.09	2.20	1.11	2.21	1.20	2.12
16	1.21	2.54	1.22	4.39	1.38	4.16	1.52	3.75
32	1.48	3.89	1.35	6.14	1.69	4.60	2.02	4.47
64	1.91	4.63	1.56	5.90	2.43	4.41	3.30	4.67
128	2.28	5.65	1.88	6.81	2.74	5.47	4.67	5.78
256	2.63	6.38	2.15	7.25	3.27	6.06	5.92	6.57
512	2.91	6.68	2.42	7.84	3.65	6.71	6.79	7.21
1024	3.66	10.56	2.90	13.04	5.22	11.54	11.91	12.46
2048	4.65	15.91	3.35	20.72	6.12	19.37	19.50	20.16
4096	5.75	23.32	3.93	33.81	7.38	30.34	32.75	33.54

Table A.20.: Synthetic domain partitioner quality characterisation with respect to problem linearity. NO is the numerical optimiser-based partitioner, while MILP is the Mixed Integer Linear Programming Partitioner.

Case	1		2		3		4	
Partitioner	ML	MILP	ML	MILP	ML	MILP	ML	MILP
ψ								
10^{-5}	1.22	1.22	1.23	1.23	1.69	2.21	-	-
10^{-4}	1.20	1.20	1.27	1.27	1.74	2.15	1.45	3.84
10^{-3}	1.20	1.21	1.28	1.30	1.64	1.97	1.52	4.48
10^{-2}	1.27	1.38	1.31	1.49	1.76	2.33	1.82	4.05
10^{-1}	1.76	2.64	2.39	3.54	2.60	4.01	3.41	5.39
1	3.45	8.85	2.53	11.77	4.37	10.54	10.88	11.83
10^{1}	3.34	15.76	2.50	20.19	4.44	16.67	15.03	23.68
10^{2}	3.38	17.81	2.74	22.39	4.66	19.65	13.20	22.46
10^{3}	3.33	17.67	2.63	23.22	4.44	20.59	15.64	27.16
10^{4}	3.32	17.69	2.63	23.23	4.40	20.45	15.64	27.18
10^{5}	3.32	17.69	2.07	17.43	4.07	19.12	14.56	25.32

A.6.3. Empirical Evaluation

The results plotted in Section 7.2.2 are reported in Table A.21 in full.

Table A.21.: Verification of partitioning approaches with real workload.

Target Accuracy	Metric	Heuristic	ML	MILP
\$1	Accuracy (\$)	0.1993	0.4819	0.4950
	Latency(s)	1204.027	6.821	4.408
\$0.5	Accuracy (\$)	0.1776	0.3353	0.3677
	Latency(s)	1204.027	6.821	4.408
\$0.1	Accuracy(\$)	0.0782	0.0856	0.0993
	Latency(s)	1211.211	38.380	6.385
\$0.05	Accuracy(\$)	0.0466	0.0480	0.0553
	Latency(s)	1204.822	50.277	7.928
\$0.01	Accuracy(\$)	0.0109	0.0101	0.0120
	Latency(s)	1471.281	244.848	27.315
\$0.005	Accuracy(\$)	0.0052	0.0052	0.0064
	Latency(s)	1700.833	477.604	66.628
\$0.001	Accuracy(\$)	0.0010	0.0010	0.0014
	Latency(s)	2896.388	2746.915	1332.630

B. Forward Financial Framework Documentation

Forward Financial Framework

Generated by Doxygen 1.8.9.1

Wed Jul 22 2015 08:31:56

Contents

Chapter 1

Main Page

ForwardFinancialFramework

F^3 is a Python-based application framework for valuing forward looking financial products on Heterogeneous Parallel Computing Platforms.

Introduction

The vision of F^3 is to allow financial engineers to express valuation computations naturally while taking advantage of the plethora of new computing platforms available.

The application framework also serves as a test case for research into domain-specific, heterogeneous computing.

Current Underlyings and Derivatives Supported:

- Black-Scholes Stochastic Underlyings
- Heston-based Stochastic Underlyings
- European Options
- European Single and Double Barrier Options
- European Double Digital Barrier Options
- European Asian Options

Platforms:

- Multicore CPUs (via GCC and Posix threads)
- Maxeler FPGA Platforms (via Maxcompiler)
- Xilinx FPGAs (via VivadoHLS)
- Altera FPGAs (via Altera's OpenCL SDK)
- GPUs and Co-Processors (via OpenCL)

In Progress:

- Derivatives with American exercise properties
- Automatic scheduling of tasks across a range of platforms

Coming Soon:

- Interest-rate derivatives
- Lattice-based Solvers

Framework Layout

"' ForwardFinancialFramework /bin - the experimental scripts for various portfolios /Derivatives - the financial derivatives classes /Platforms - the platform classes /Solvers - the solver alogrithms /Underlyings - the underlyings classes "'

Installation

F^3 requires:

- A ∗nix-based Operating System
- `GCC`
- `Python` >= 2.7
- `Numpy`
- Maxcompiler version 12.2 >= (For Maxeler code)
- `PyOpenCL` (for any OpenCL Execution)
- X OpenCL SDK (where X is the vendor of the OpenCL platform in question)
- Xilinx Vivado HLS 2013.4 (For VivadoHLS code)

The following environmental variables also need to be set:

- `F3_ROOT` needs to be equal to the location of this repository, including the directory name (e.↩ g. `/home/[Username]/workspace/ForwardFinancialFramework`)
- `PYTHONPATH=$PYTHONPATH:$F3_ROOT/..`

Getting Started

1. Change to the test_script directory (i.e. ForwardFinancialFramework/bin/test_scripts)
2. Run the following command in the script directory: `python \<script file name\> script options` e.g. `python mc_solver_test.py CPU Execute` would run a very basic, CPU-based bond valuation.

Extending the Framework

- To add a new derivative or underlying, look at the existing derivatives and underlyings as an example. The basic procedure:
  1. Create a new class in the correct directory, inheriting from Option.py or Underlying.py respectively.
  2. Overload or add the required methods and variables for the solver(s) being targetted to the new class being created.
  3. Create the required supporting libraries for generating the platform-solver code.
- To add a new solver or platform, its a bit more involved. Again, look at the existing ones for ideas.

Contact Info

Please, feel free to get in touch with me (gordon.inggs (at) gmail.com). I'm particularly happy to provide comparison data for your option evaluations.

Chapter 2

Namespace Index

2.1 Packages

Here are the packages with brief descriptions (if available):

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

Namespace Documentation

5.1 ForwardFinancialFramework Namespace Reference

5.1.1 Detailed Description

```
@package ForwardFinancialFramework
Root module for ForwardFinancialFramework

Deliberately empty
```

5.2 ForwardFinancialFramework.Derivatives.Asian_Option Namespace Reference

Classes

- class Asian_Option

5.2.1 Detailed Description

```
Created on 17 June 2012
```

5.3 ForwardFinancialFramework.Derivatives.Barrier_Option Namespace Reference

Classes

- class Barrier_Option

5.3.1 Detailed Description

```
Created on 16 June 2012
```

5.4 ForwardFinancialFramework.Derivatives.Digital_Double_Barrier_Option Namespace Reference

Classes

- class Digital_Double_Barrier_Option

5.4.1 Detailed Description

```
Created on 17 June 2012
```

5.5 ForwardFinancialFramework.Derivatives.Double_Barrier_Option Namespace Reference

Classes

- class Double_Barrier_Option

5.5.1 Detailed Description

```
Created on 17 June 2012
```

5.6 ForwardFinancialFramework.Derivatives.European_Option Namespace Reference

Classes

- class European_Option

5.6.1 Detailed Description

```
Created on 30 May 2012
```

5.7 ForwardFinancialFramework.Derivatives.Option Namespace Reference

Classes

- class Option

5.7.1 Detailed Description

```
@package ForwardFinancialFramework.Derivatives.Option

This package contains the base derivative class, option.
Created on 30 May 2012
```

5.8 ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA Namespace Reference

Classes

- class MaxelerFPGA

5.8.1 Detailed Description

```
Created on 26 October 2012
```

5.9 ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo Namespace Reference

Classes

- class MaxelerFPGA_MonteCarlo

5.9.1 Detailed Description

```
Created on 30 October 2012
```

5.10 ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU Namespace Reference

Classes

- class MulticoreCPU

5.10.1 Detailed Description

```
Created on 11 July 2012
```

5.11 ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo Namespace Reference

Classes

- class MulticoreCPU_MonteCarlo

5.11.1 Detailed Description

```
Created on 30 October 2012
```

5.12 ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPG↩A Namespace Reference

Classes

- class OpenCLAlteraFPGA

5.12.1 Detailed Description

```
Created on 1 April 2014
```

5.13 ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_↩ MonteCarlo Namespace Reference

Classes

- class OpenCLAlteraFPGA_MonteCarlo

5.13.1 Detailed Description

```
Created on 1 April 2014
```

5.14 ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU Namespace Reference

Classes

- class OpenCLGPU

5.14.1 Detailed Description

```
Created on 23 February 2013
```

5.15 ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU_MonteCarlo Namespace Reference

Classes

- class OpenCLGPU_MonteCarlo

5.15.1 Detailed Description

```
Created on 23 February 2013
```

5.16 ForwardFinancialFramework.Platforms.Platform Namespace Reference

Classes

- class Platform

5.16.1 Detailed Description

```
@package ForwardFinancialFramework.Platforms.Platform

This package contains the base platform class
Created on 23 November 2014
```

5.17 ForwardFinancialFramework.Solvers.MonteCarlo.MonteCarlo Namespace Reference

Classes

- class MonteCarlo

5.17.1 Detailed Description

```
@package ForwardFinancialFramework.Solvers.MonteCarlo

This is the base class for all Monte Carlo solvers.
Created on 11 July 2012
```

5.18 ForwardFinancialFramework.Underlyings.Black_Scholes Namespace Reference

Classes

- class Black_Scholes

5.18.1 Detailed Description

```
@package ForwardFinancialFramework.Underlyings

Created on 30 May 2012
```

5.19 ForwardFinancialFramework.Underlyings.Heston Namespace Reference

Classes

- class Heston

5.19.1 Detailed Description

```
@package ForwardFinancialFramework.Underlyings

Created on 12 June 2012
```

5.20 ForwardFinancialFramework.Underlyings.Underlying Namespace Reference

Classes

- class Underlying

5.20.1 Detailed Description

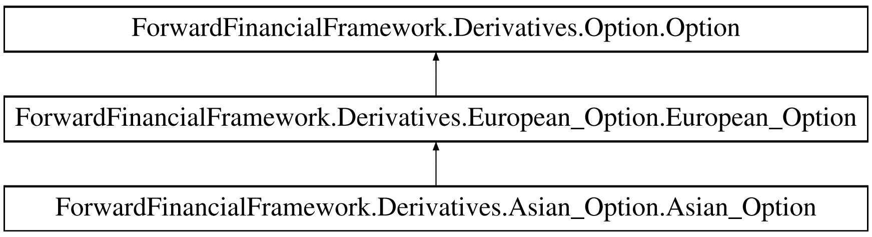
```
@package ForwardFinancialFramework.Underlyings

This package contains the underlying classes.
Created on 30 May 2012
```

Chapter 6

Class Documentation

6.1 ForwardFinancialFramework.Derivatives.Asian_Option.Asian_Option Class Reference

Inheritance diagram for ForwardFinancialFramework.Derivatives.Asian_Option.Asian_Option:


ForwardFinancialFramework.Derivatives.Option.Option
ForwardFinancialFramework.Derivatives.European_Option.European_Option
ForwardFinancialFramework.Derivatives.Asian_Option.Asian_Option

Public Member Functions

- def __init__ (self, underlying, time_period, call, strike_price, points)
- def path_init (self)
- def path (self, price, time)
- def payoff (self, end_price)

Public Attributes

- **points**
- **average_value**
- **delta_time**

Static Public Attributes

- string **name** = "asian_option"
- int points = 0

  Number of points over which to find the average price.
- float average_value = 0.0

  Accumulator variable for calculating the average underlying value.

6.1.1 Detailed Description

```
Asian Option Class

This class represents an arithmetic Asian Option derivative product.
```

6.1.2 Constructor & Destructor Documentation

6.1.2.1 def ForwardFinancialFramework.Derivatives.Asian_Option.Asian_Option.__init__ (*self, underlying, time_period, call, strike_price, points*)

```
Constructor

Parameters
    underlying, time_period, call, strike_price, points - same as for European_Option
    points - number of points, evenly spaced over lifetime overwhich to take spot price average value
```

6.1.3 Member Function Documentation

6.1.3.1 def ForwardFinancialFramework.Derivatives.Asian_Option.Asian_Option.path (*self, price, time*)

```
Path evolution method

Parameters
    price - (float) the current value of the underlying
    time - (float) the current time of the underlying
```

6.1.3.2 def ForwardFinancialFramework.Derivatives.Asian_Option.Asian_Option.path_init (*self*)

```
Path initialisation method

Parameters
    None
```

6.1.3.3 def ForwardFinancialFramework.Derivatives.Asian_Option.Asian_Option.payoff (*self, end_price*)

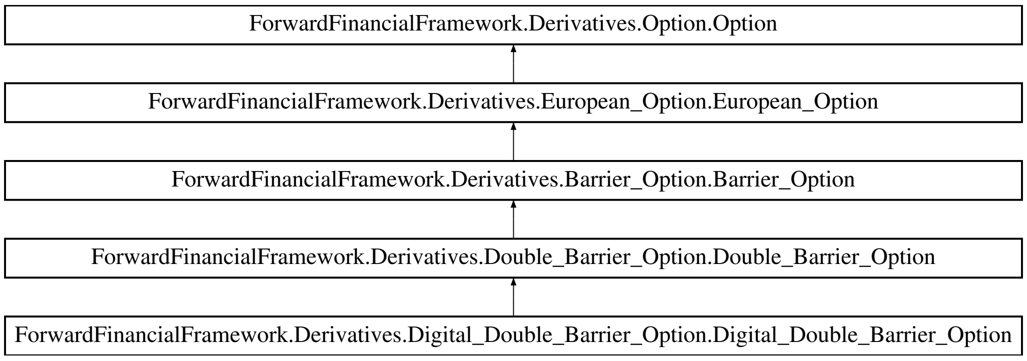
```
Payoff evaluation method

Parameters
    end_price - (float) the end price of the underlying
```

The documentation for this class was generated from the following file:

- Asian_Option.py

6.2 ForwardFinancialFramework.Derivatives.Barrier_Option.Barrier_Option Class Reference

Inheritance diagram for ForwardFinancialFramework.Derivatives.Barrier_Option.Barrier_Option:


ForwardFinancialFramework.Derivatives.Option.Option
ForwardFinancialFramework.Derivatives.European_Option.European_Option
ForwardFinancialFramework.Derivatives.Barrier_Option.Barrier_Option
ForwardFinancialFramework.Derivatives.Double_Barrier_Option.Double_Barrier_Option
ForwardFinancialFramework.Derivatives.Digital_Double_Barrier_Option.Digital_Double_Barrier_Option

Public Member Functions

- def __init__ (self, underlying, time_period, call, strike_price, points, barrier, out, down)
- def path (self, price, time)
- def payoff (self, end_price)

Public Attributes

- **points**
- **barrier**
- **delta_time**
- **value**

Static Public Attributes

- string **name** = "barrier_option"
- float barrier = 0.0

  Price Barrier.
- out = None

  Out barrier indication.
- down = None

  Down barrier indication.
- int points = 0

  Number of barrier points to check.
- **barrier_event** = None

6.2.1 Detailed Description

```
Barrier Option Class

This class representes barrier option products
```

6.2.2 Constructor & Destructor Documentation

6.2.2.1 def ForwardFinancialFramework.Derivatives.Barrier_Option.Barrier_Option.__init__ (*self, underlying, time_period, call, strike_price, points, barrier, out, down*)

```
Constructor

        Parameters
underlying, time_period, call, strike_price - same as European Option
```

```
points - (int) number of points at which to check barrier, spaced evenly over option lifetime
barrier - (float) price value of the barrier
out - (bool) is this an out barrier?
down - (bool) is this a down barrier (i.e. has to cross from barrier from above)
```

6.2.3 Member Function Documentation

6.2.3.1 def ForwardFinancialFramework.Derivatives.Barrier_Option.Barrier_Option.path (*self*, *price*, *time*)

```
Path evolution method

Parameters
    price - (float) current spot price of underlying
    time - (float) current time of underlying
```

6.2.3.2 def ForwardFinancialFramework.Derivatives.Barrier_Option.Barrier_Option.payoff (*self*, *end_price*)

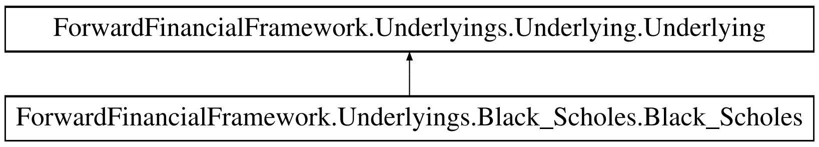
```
Payoff method

Parameters
    end_price - (float) final price of underlying asset
```

The documentation for this class was generated from the following file:

- Barrier_Option.py

6.3 ForwardFinancialFramework.Underlyings.Black_Scholes.Black_Scholes Class Reference

Inheritance diagram for ForwardFinancialFramework.Underlyings.Black_Scholes.Black_Scholes:


ForwardFinancialFramework.Underlyings.Underlying.Underlying
ForwardFinancialFramework.Underlyings.Black_Scholes.Black_Scholes

Public Member Functions

- def __init__ (self, rfir, current_price, volatility)
- def **__repr__** (self)

Public Attributes

- **volatility**

Static Public Attributes

- string **name** = "black_scholes_underlying"
- float volatility = 0.0

  The constant volatility of the product.

6.3.1 Detailed Description

```
Black Scholes Underlying class

This class represents a Black Scholes model based underlying. It inheirts from the Underlying base class.
```

6.3.2 Constructor & Destructor Documentation

6.3.2.1 def ForwardFinancialFramework.Underlyings.Black_Scholes.Black_Scholes.__init__ (*self*, *rfir*, *current_price*, *volatility*)

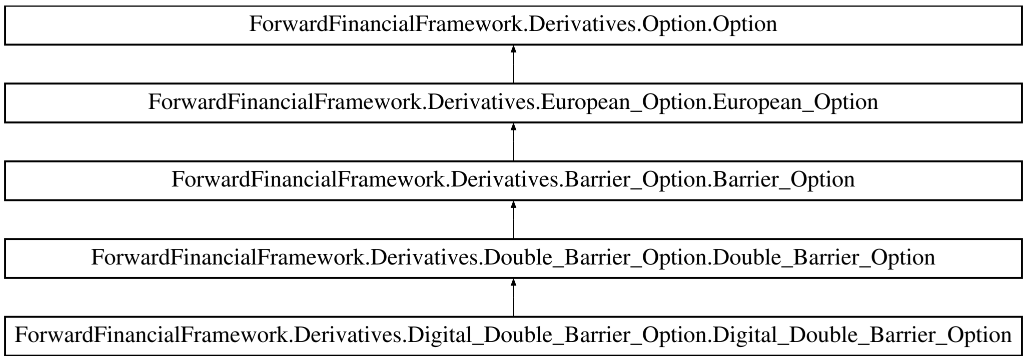
```
Constructor

Parameters
    rfir and current price same as Underlying
volatility - (float) size of constant volatility
```

The documentation for this class was generated from the following file:

- Black_Scholes.py

6.4 ForwardFinancialFramework.Derivatives.Digital_Double_Barrier_Option.Digital_↩ Double_Barrier_Option Class Reference

Inheritance diagram for ForwardFinancialFramework.Derivatives.Digital_Double_Barrier_Option.Digital_Double_↩ Barrier_Option:


ForwardFinancialFramework.Derivatives.Option.Option
ForwardFinancialFramework.Derivatives.European_Option.European_Option
ForwardFinancialFramework.Derivatives.Barrier_Option.Barrier_Option
ForwardFinancialFramework.Derivatives.Double_Barrier_Option.Double_Barrier_Option
ForwardFinancialFramework.Derivatives.Digital_Double_Barrier_Option.Digital_Double_Barrier_Option

Public Member Functions

- def __init__ (self, underlying, time_period, call, strike_price, points, barrier, out, down, second_barrier)

Public Attributes

- **value**

Static Public Attributes

- string **name** = "digital_double_barrier_option"

6.4.1 Detailed Description

```
Digital Double Barrier Class

This class represents a digital double barrier derivative product.
```

6.4.2 Constructor & Destructor Documentation

6.4.2.1 def ForwardFinancialFramework.Derivatives.Digital_Double_Barrier_Option.Digital_Double_Barrier_Option.__init__ (*self, underlying, time_period, call, strike_price, points, barrier, out, down, second_barrier*)

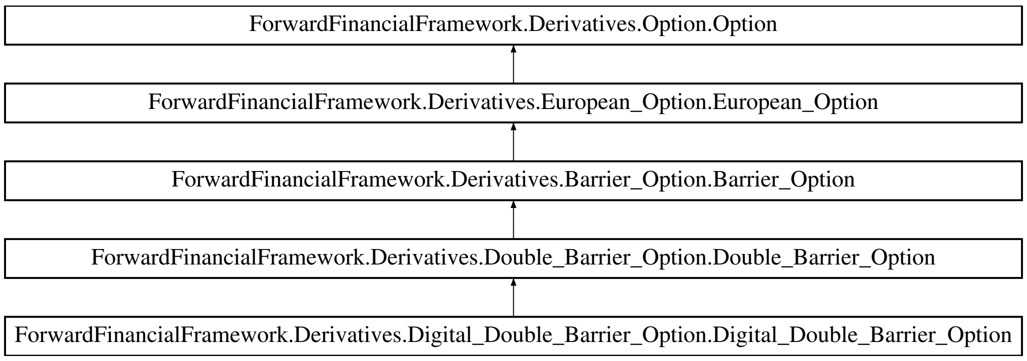
```
Constructor

        Parameters
underlying, time_period, call, strike_price, points, barrier, out, down, second_barrier - same as double barri
```

The documentation for this class was generated from the following file:

- Digital_Double_Barrier_Option.py

6.5 ForwardFinancialFramework.Derivatives.Double_Barrier_Option.Double_Barrier_↵ Option Class Reference

Inheritance diagram for ForwardFinancialFramework.Derivatives.Double_Barrier_Option.Double_Barrier_Option:


ForwardFinancialFramework.Derivatives.Option.Option
ForwardFinancialFramework.Derivatives.European_Option.European_Option
ForwardFinancialFramework.Derivatives.Barrier_Option.Barrier_Option
ForwardFinancialFramework.Derivatives.Double_Barrier_Option.Double_Barrier_Option
ForwardFinancialFramework.Derivatives.Digital_Double_Barrier_Option.Digital_Double_Barrier_Option

Public Member Functions

- def __init__ (self, underlying, time_period, call, strike_price, points, barrier, out, down, second_barrier)
- def **__repr__** (self)

Public Attributes

- **second_barrier**
- **barrier_event**

Static Public Attributes

- string **name** = "double_barrier_option"
- float second_barrier = 0.0

  Second price barrier.

- float down = 1.0

  By default the this option is a down barrier is now true, as the double barrier is between two points.

6.5.1 Detailed Description

```
Double Barrier Option class

This class represents a double barrier option deriviative production.
```

6.5.2 Constructor & Destructor Documentation

6.5.2.1 def ForwardFinancialFramework.Derivatives.Double_Barrier_Option.Double_Barrier_Option.__init__ (*self, underlying, time_period, call, strike_price, points, barrier, out, down, second_barrier*)

```
Constructor
Parameters
    underlying,time_period,call,strike_price,points,barrier,out,down - same as Barrier Option
    second_barrier - (float) the second price barrier
```

6.5.3 Member Data Documentation

6.5.3.1 float ForwardFinancialFramework.Derivatives.Double_Barrier_Option.Double_Barrier_Option.down = 1.0 [static]

By default the this option is a down barrier is now true, as the double barrier is between two points.

By enforced convention, the first is the lower barrier

6.5.3.2 float ForwardFinancialFramework.Derivatives.Double_Barrier_Option.Double_Barrier_Option.second_barrier = 0.0 [static]

Second price barrier.

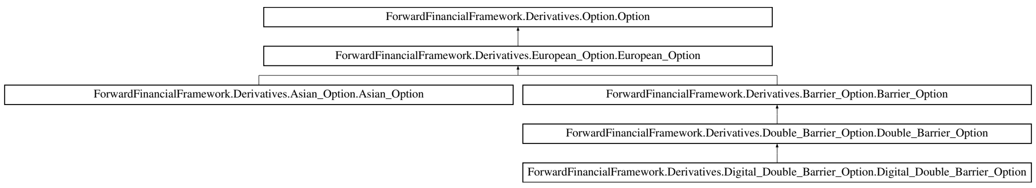
By definition, this is the higher price barrier

The documentation for this class was generated from the following file:

- Double_Barrier_Option.py

6.6 ForwardFinancialFramework.Derivatives.European_Option.European_Option Class Reference

Inheritance diagram for ForwardFinancialFramework.Derivatives.European_Option.European_Option:


ForwardFinancialFramework.Derivatives.Option.Option
ForwardFinancialFramework.Derivatives.European_Option.European_Option
ForwardFinancialFramework.Derivatives.Asian_Option.Asian_Option
ForwardFinancialFramework.Derivatives.Barrier_Option.Barrier_Option
ForwardFinancialFramework.Derivatives.Double_Barrier_Option.Double_Barrier_Option
ForwardFinancialFramework.Derivatives.Digital_Double_Barrier_Option.Digital_Double_Barrier_Option

Public Member Functions

- def __init__ (self, underlying, time_period, call, strike_price)

Public Attributes

- **value**

Static Public Attributes

- string **name** = "european_option"

6.6.1 Detailed Description

```
European Option class

This class represents a European or Vanilla option pricing product
```

6.6.2 Constructor & Destructor Documentation

6.6.2.1 def ForwardFinancialFramework.Derivatives.European_Option.European_Option.__init__ (*self, underlying, time_period, call, strike_price*)

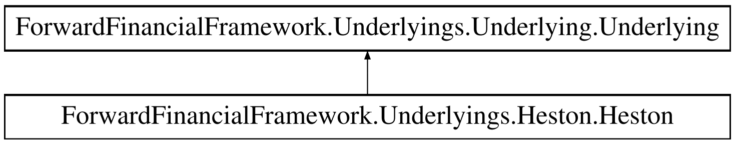
```
Constructor

Parameters
    underlying, time_period, call, strike_price - same as for Option.Option
```

The documentation for this class was generated from the following file:

- European_Option.py

6.7 ForwardFinancialFramework.Underlyings.Heston.Heston Class Reference

Inheritance diagram for ForwardFinancialFramework.Underlyings.Heston.Heston:


ForwardFinancialFramework.Underlyings.Underlying.Underlying
ForwardFinancialFramework.Underlyings.Heston.Heston

Public Member Functions

- def __init__
- def path (self, delta_time)
- def **__repr__** (self)

Public Attributes

- **initial_volatility**
- **volatility_volatility**
- **rho**
- **kappa**
- **theta**
- **volatility**

- **correlation_matrix_1_1**
- **correlation_matrix_0_0**
- **correlation_matrix_0_1**
- **correlation_matrix_1_0**

Static Public Attributes

- string **name** = "heston_underlying"
- float initial_volatility = 0.0

  Starting value for the volatility.
- float volatility_volatility = 0.0

  The constant volatility of the volatility.
- float rho = 0.0

  correlation factor between evolution of price and volatility
- float kappa = 0.0

  volatility evolution reversion rate
- float theta = 0.0

  long run mean of volatility evolution, analogous to the rfir for the underlying price
- float correlation_matrix_0_0 = 0.0

  Attributes storing the Cholesky matrix of the correlation between the two random numbers generated.
- float **correlation_matrix_0_1** = 0.0
- float **correlation_matrix_1_0** = 0.0
- float **correlation_matrix_1_1** = 0.0
- float volatility = 0.0

  Volatility is a variable in this instance.

6.7.1 Detailed Description

```
Heston Model Underlying class
```

6.7.2 Constructor & Destructor Documentation

6.7.2.1 def ForwardFinancialFramework.Underlyings.Heston.Heston.__init__ (*self*, *rfir*, *current_price*, *initial_volatility*, *volatility_volatility*, *rho*, *kappa*, *theta*, *correlation_matrix_0_0* = None, *correlation_matrix_0_1* = None, *correlation_matrix_1_0* = None, *correlation_matrix_1_1* = None)

```
Constructor

Parameters
    rfir,current_price - same as for Underlying.Underlying
    initial_volatility - (float) initial volatility value
    volatility_volatility - (float) the constnat volatility of the volatility
    rho - (float) the correlation factor between the price and volatility evolution
    kappa - (float) the volatility evolution reversion rate
    theta - (float) the long run mean of the volatility evolution
```

6.7.3 Member Function Documentation

6.7.3.1 def ForwardFinancialFramework.Underlyings.Heston.Heston.path (*self*, *delta_time*)

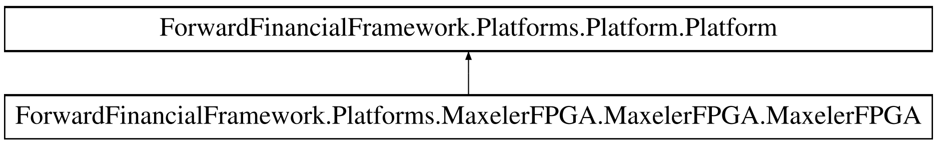
```
Path evolution method

Parameters
    delta_time - (float) the time step by which the price should be evoloved.
```

The documentation for this class was generated from the following file:

- Heston.py

6.8 ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA.MaxelerFPG↩ A Class Reference

Inheritance diagram for ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA.MaxelerFPGA:


ForwardFinancialFramework.Platforms.Platform.Platform
ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA.MaxelerFPGA

Public Member Functions

- def __init__
- def __del__ (self)

Public Attributes

- board

  The name of Maxeler board to use.
- device_resources

  The integer resource units available.
- clock_rate

  The integer clock rate in Megahertz to use during the build process.
- **boardid**

Static Public Attributes

- string **name** = "maxeler_fpga"
- int **threads** = 1

6.8.1 Detailed Description

```
Maxeler FPGA Platform Class

This class is for representing Maxeler FPGAs. If a Max4 FPGA is being used, it is assumed that the Max orchist
```

6.8.2 Constructor & Destructor Documentation

6.8.2.1 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA.MaxelerFPGA.__init__ (*self*, *platform_directory_string* = "Platforms/**MaxelerFPGA**/maxeler_code/build", *root_directory_string* = None, *ssh_alias* = "", *remote* = False, *hostname* = "", *shell_vars* = {}, *board* = "max3", *boardid* = ":0")

```
Constructor

Parameters
```

```
    platform_directory_string,root_directory_string,ssh_alias,remote,hostname,shell_vars - same as for the Pla
    board - (string) Maxeler board to use
    boardid - (string) ID of Maxeler board to use
```

6.8.2.2 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA.MaxelerFPGA.__del__ (*self*)

```
Deconstructor
unreserves the board from the Maxorchestrator
```

6.8.3 Member Data Documentation

6.8.3.1 ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA.MaxelerFPGA.board

The name of Maxeler board to use.

Can either be Max3 or Max4

6.8.3.2 ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA.MaxelerFPGA.clock_rate

The integer clock rate in Megahertz to use during the build process.

6.8.3.3 ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA.MaxelerFPGA.device_resources

The integer resource units available.

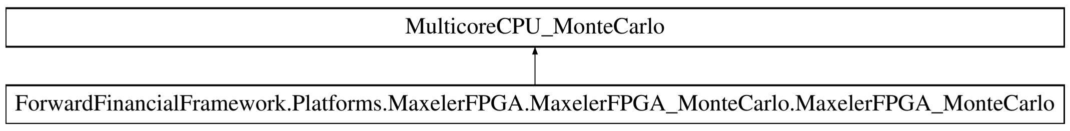
This is used by the Maxeler solver class.

The documentation for this class was generated from the following file:

- MaxelerFPGA.py

6.9 ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.↩ MaxelerFPGA_MonteCarlo Class Reference

Inheritance diagram for ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.↩
MaxelerFPGA_MonteCarlo:


MulticoreCPU_MonteCarlo
ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo

Public Member Functions

- def __init__
- def set_default_parameters (self)
- def generate_name (self)
- def **generate_identifier** (self)
- def generate
- def generate_activity_thread (self)
- def generate_libraries (self)
- def generate_kernel

- def generate_manager (self)
- def compile
- def dummy_run (self)
- def get_delay (self)

Public Attributes

- pipelining

    Integer degree of loop unrolling to be performed.
- instances

    Number of parallel task instances to be performed.
- **delay**
- **activity_thread_name**

Static Public Attributes

- c_slow = False

    Option for whether c-slowing optimisation should be used.
- int delay = 10

    Integer delay value to be used if the c-slowing optimisation is not used.
- int **pipelining** = 1
- int **instances** = 1

6.9.1 Detailed Description

```
Maxeler Monte Carlo solver class

This class provides the generation and compilation behaviour for the Maxeler FPGA platform.
```

6.9.2 Constructor & Destructor Documentation

6.9.2.1 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.__init_↩_ (*self*, *derivative*, *paths*, *platform*, *points* = `4096`, *reduce_underlyings* = `True`, *instance_paths* = `None`, *c_slow* = `None`, *pipelining* = `None`, *instances* = `None`)

```
Constructor

Parameters
    derivative, paths, platform, reduce_underlyings - same as for MulticoreCPU_MonteCarlo class
    instance_paths - (int) number of paths to perform per call to the Maxeler DFE
    c_slow - (bool) option to use c-slowing optimisation
    pipelining - (int) amount of loop unrolling to perform
    instances - (int) number of parallel instances to use
```

6.9.3 Member Function Documentation

6.9.3.1 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.compile (*self*, *override* = `True`, *cleanup* = `True`, *debug* = `True`)

```
Compiler method override for Maxeler solvers.

Makes use of GNU make infrastructure underneath.

Parameters
    override, cleanup, debug - same as in other solver classes
```

6.9.3.2 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.↩ dummy_run (*self*)

```
Helper method for wiping the configuration of the current Maxeler board.
```

6.9.3.3 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.generate (*self*, *override* = True, *verbose* = False, *debug* = False)

```
Overriding generate method. In addition to host code, the code for Maxeler DFE and its manager class are gener

Parameters
    override, verbose, debug - same as for MulticoreCPU_MonteCarlo class
```

6.9.3.4 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.↩ generate_activity_thread (*self*)

```
Overriding the generate activity thread method so that it sets up and communicates with the Maxeler DFE
```

6.9.3.5 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.↩ generate_kernel (*self*, *overide* = True)

```
Helper method for generating the kernel.

Parameters
    overide - (bool) Force the code to be generated.
```

6.9.3.6 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.↩ generate_libraries (*self*)

```
Overriding the libraries generation
```

6.9.3.7 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.↩ generate_manager (*self*)

```
Helper method for generating Maxeler hardware manager, which specifies communication and various build properi
```

6.9.3.8 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.↩ generate_name (*self*)

```
Overriding helper method to include board parameters
```

6.9.3.9 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.get_↩ delay (*self*)

```
Helper method for finding delay required
```

6.9.3.10 def ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA_MonteCarlo.MaxelerFPGA_MonteCarlo.set_↩ default_parameters (*self*)

```
Helper method for setting the default FPGA parameters to use
```

The documentation for this class was generated from the following file:

- MaxelerFPGA_MonteCarlo.py

6.10 ForwardFinancialFramework.Solvers.MonteCarlo.MonteCarlo.MonteCarlo Class Reference

Public Member Functions

- def __init__
- def generate
- def compile (self)
- def execute
- def cleanup (self)
- def setup_underlyings (self, reduce_underlyings)
- def generate_name (self)
- def attribute_stripper (self, attributes, variables)

Public Attributes

- **solver_metadata**
- **derivative**
- **underlying**
- **underlying_dependencies**
- **underlying_attributes**
- **underlying_variables**
- **derivative_attributes**
- **derivative_variables**
- **output_file_name**

Static Public Attributes

- string **name** = "monte_carlo_solver"
- paths = None

  Monte Carlo simulation paths.
- threads = None

  Number of threads of execution to use.
- reduce_underlyings = None

  Fusion optimisation.
- platform = None

  Platform of exectuion.
- list derivative = [ ]

  Derivative products to value.
- list derivative_attributes = [ ]

  Derivative product attributes.
- list derivative_variables = [ ]

Derivative product variables.

- list underlying = [ ]

  Underlyings of derivative products.
- list underlying_attributes = [ ]

  Underlying attributes.
- list underlying_variables = [ ]

  Underlying variables.
- list **underlying_dependencies** = [ ]

6.10.1 Detailed Description

```
Base Monte Carlo solver class

This class is the base class for all of the Monte Carlo solvers
```

6.10.2 Constructor & Destructor Documentation

6.10.2.1 def ForwardFinancialFramework.Solvers.MonteCarlo.MonteCarlo.MonteCarlo.__init__ (*self*, *derivative*, *paths*, *platform*, *reduce_underlyings* = True)

```
Constructor

Parameters
    derivatives - (list of ForwardFinancialFramework.Derivatives) list of derivative products that need to be
    paths - (int) number of Monte Carlo simulations to use
    platform - (ForwardFinancialFramework.Platform) platform to perform solving upon
    reduce_underlyings - (bool) use the fusion optimisation?
```

6.10.3 Member Function Documentation

6.10.3.1 def ForwardFinancialFramework.Solvers.MonteCarlo.MonteCarlo.MonteCarlo.attribute_stripper (*self*, *attributes*, *variables*)

```
Helper Method used to remove all items in the first list from the second list, if present

        Parameters
attributes - (list) elements to remove
variables - (list) list to remove from

        returns variables - attributes
```

6.10.3.2 def ForwardFinancialFramework.Solvers.MonteCarlo.MonteCarlo.MonteCarlo.cleanup (*self*)

```
Method for cleaning up solver
```

6.10.3.3 def ForwardFinancialFramework.Solvers.MonteCarlo.MonteCarlo.MonteCarlo.compile (*self*)

```
Method for compiling generated solver code
```

6.10.3.4 def ForwardFinancialFramework.Solvers.MonteCarlo.MonteCarlo.MonteCarlo.execute (*self*, *cleanup* = False)

```
Method for running solver on specified platform
```

6.10.3.5 def ForwardFinancialFramework.Solvers.MonteCarlo.MonteCarlo.MonteCarlo.generate (*self*, *override* = `True`)

```
Method for generating solver code.

Parameters
    override - (bool) overwrite pre-existing code
```

6.10.3.6 def ForwardFinancialFramework.Solvers.MonteCarlo.MonteCarlo.MonteCarlo.generate_name (*self*)

```
Method for generating unique name for solver, based upon underlying and derivative produces
```

6.10.3.7 def ForwardFinancialFramework.Solvers.MonteCarlo.MonteCarlo.MonteCarlo.setup_underlyings (*self*, *reduce_underlyings*)

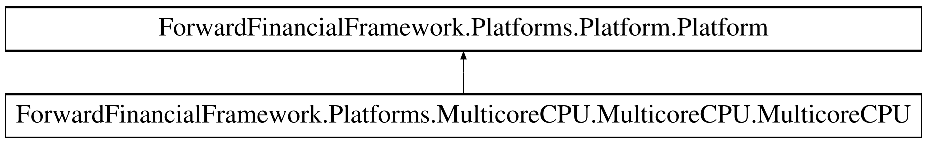
```
utility method for generating list of underlyings from the solver's derivatives

Parameters
    reduce_underlysin - (bool) apply fusion optimisations?
```

The documentation for this class was generated from the following file:

- MonteCarlo.py

6.11 ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU.MulticoreCPU Class Reference

Inheritance diagram for ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU.MulticoreCPU:


ForwardFinancialFramework.Platforms.Platform.Platform
ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU.MulticoreCPU

Public Member Functions

- def __init__

Public Attributes

- **threads**

Static Public Attributes

- string **name** = "multicore_cpu"
- int threads = 1

  Number of computational threads being used on this platform.

6.11.1 Detailed Description

```
Multicore CPU Platform Class
```

6.11.2 Constructor & Destructor Documentation

6.11.2.1 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU.MulticoreCPU.__init__ (*self*, *threads* = None, *platform_directory_string* = "Platforms/**MulticoreCPU**/multicore_c_code", *root_directory_string* = None, *ssh_alias* = "", *remote* = False, *hostname* = None)

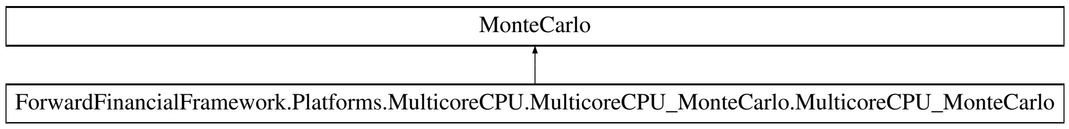
```
Constructor

        Parameters
platform_directory_string, root_directory_String, ssh_alias, remote, hostname - same as Platform class
threads - (int) number of computational threads to use. If not set, the number of cores on the machine will be
```

The documentation for this class was generated from the following file:

- MulticoreCPU.py

6.12 ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.↩ MulticoreCPU_MonteCarlo Class Reference

Inheritance diagram for ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.↩ MulticoreCPU_MonteCarlo:


MonteCarlo
ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo

Public Member Functions

- def __init__
- def generate
- def generate_identifier (self)
- def generate_libraries (self)
- def generate_variable_declaration (self)
- def generate_main_thread (self)
- def generate_activity_thread_unpacking (self)
- def generate_underlying_derivative_path_initialisations
- def generate_activity_thread (self)
- def compile
- def execute
- def generate_source
- def generate_base_class_names (self, tempclass, templist)

Public Attributes

- **utility_libraries**
- **activity_thread_name**
- **header_string**
- **random_number_generator**

Static Public Attributes

- floating_point_format = None
  Format used for floating point computations.

6.12.1 Detailed Description

```
MulticoreCPU Monte Carlo class

This class is for generating, compiling and executing Monte Carlo solvers upon Multicore CPU platforms. Many c
```

6.12.2 Constructor & Destructor Documentation

6.12.2.1 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.↩ __init__ (*self,* *derivative,* *paths,* *platform,* *reduce_underlyings* = True, *random_number_generator* = "taus_ziggurat", *floating_point_format* = "float", *default_points* = 4096)

```
Constructor

Parameters
    derivative - (list of Derivative.Option) derivative products to be priced
    paths - (int) number of simulation paths to use. Only used for execution behaviour
    platform - (Platfrom.MulticoreCPU.MulticoreCPU) Multicore CPU platform to use
    reduce_underlyings - (bool) optimisation option, collapse identical underlyings together.
    random_number_generator - (string) Gaussian random number generator to use. Valid values include "taus_zig
    floating_point_format - (string) Floating point standard to use. Acceptable values include "float","double
    default_points - (int) Default number of discretisation points to use in a simulation, unless otherwise sp
```

6.12.3 Member Function Documentation

6.12.3.1 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.compile (*self,* *overide* = True, *compile_options* = [], *debug* = False, *profile* = False)

```
Compile method

        This compiles the generated source code.

        Parameters
override - (bool) option to force the compilation
compile_options - (list of strings) pass in any compiler options
debug - (bool) option to compile with debugging symbols
profile - (bool) option to compile with profiling symbols *big performance hit*
```

6.12.3.2 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.execute (*self,* *debug* = False, *seed* = None, *timeout* = None)

```
Execute method. This runs the generated and compiled solver (assuming it exists).

This method is reused by many of the Class's children

Parameters
    debug - (bool) option to increase verbosity, including the binary file and its parameters
    seed - (int) value to seed the solver with.
    timeout - (int) timeout in seconds to wait before killing the solver
```

6.12.3.3 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_Monte↩ Carlo.generate (*self,* *name_extension* = ".c", *override* = True, *verbose* = False, *debug* = False)

```
Code generation method
```

```
Parameters
    name_extension - (string) file name extension to use for generated code
    override - (bool) option to force code generation
    verbose - (bool) option for setting verbosity level of code generation
    debug - (bool) option passed to source code generation method
```

6.12.3.4 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.↩ generate_activity_thread (*self*)

```
Helper method for generating activity thread

This is the method overrided by the children of this class.
```

6.12.3.5 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.↩ generate_activity_thread_unpacking (*self*)

```
Helper method for generating active thread
```

6.12.3.6 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.↩ generate_base_class_names (*self,* *tempclass,* *templist*)

```
Helper method for pulling in various super classes during compilation
```

6.12.3.7 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.↩ generate_identifier (*self*)

```
Helper method for generating identifiers and predefines for source code

Parameters
    None
```

6.12.3.8 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.↩ generate_libraries (*self*)

```
Helper method for generating library includes
```

6.12.3.9 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.↩ generate_main_thread (*self*)

```
Helper method for generating main function

This is also used by many of the inheiriting classes
```

6.12.3.10 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.↩ generate_source (*self,* *code_string,* *name_extension =* `".c"`, *verbose =* `False`, *debug =* `False`)

```
Helper method for generating source code files

        Parameters
code_string - (list of strings) the code to be written to the file. Each entry is a new line
name_extension - (string) file extension to use on the file
verbose - (bool) option to generate verbose code
debug - (bool) option to print names of files generated
```

6.12.3.11 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_↩ MonteCarlo.generate_underlying_derivative_path_initialisations (*self*, *linking_variables* = `True`)

```
Helper method for generating underlying and derivative path initilisation behaviour
```

6.12.3.12 def ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU_MonteCarlo.MulticoreCPU_MonteCarlo.↩ generate_variable_declaration (*self*)

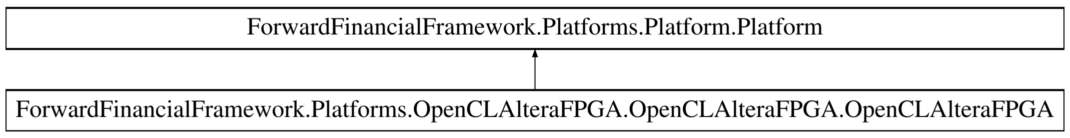
```
Helper method for generating Intermediate and Communication Variables
```

The documentation for this class was generated from the following file:

- MulticoreCPU_MonteCarlo.py

6.13 ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA.↩ OpenCLAlteraFPGA Class Reference

Inheritance diagram for ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA.OpenC↩ LAlteraFPGA:


ForwardFinancialFramework.Platforms.Platform.Platform
ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA.OpenCLAlteraFPGA

Public Member Functions

- def __init__

Public Attributes

- **platform_name**
- **board**

Static Public Attributes

- string **name** = "opencl_alterafpga"
- int threads = 0

  This variable isn't used here, so set to 0 for safety.
- device_type = pyopencl.device_type.ALL

  Currently unused, but if it was, would use ALL device type.
- string board = ""

  OpenCL Altera SDK board support package to use.

6.13.1 Detailed Description

```
OpenCL Altera FPGA Platform Class

TODO: inheirt from OpenCLGPU class, similar to the OpenCL Altera Monte Carlo solver class
```

6.13.2 Constructor & Destructor Documentation

6.13.2.1 def ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA.OpenCLAlteraFPGA.__init__ (*self*, *platform_directory_string* = "Platforms/**OpenCLAlteraFPGA**/openclalterafpga_code", *root_directory_string* = None, *platform_name* = "Altera Corporation", *board* = "pcie385n_d5", *ssh_alias* = "", *remote* = False, *hostname* = "")

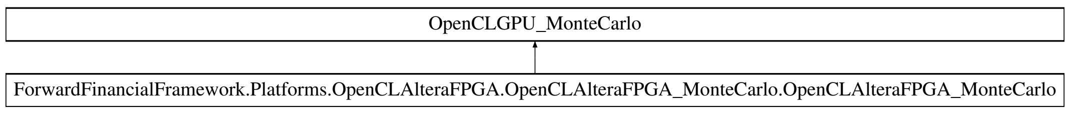
```
Constructor

        Parameters
platform_directory_string, root_directory_String, ssh_alias, remote, hostname - same as Platform class
board - (string) OpenCL Altera Certified board to use. The correct name can be found from "aoc --list-boards"
```

The documentation for this class was generated from the following file:

- OpenCLAlteraFPGA.py

6.14 ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_↩ MonteCarlo.OpenCLAlteraFPGA_MonteCarlo Class Reference

Inheritance diagram for ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_Monte↩
Carlo.OpenCLAlteraFPGA_MonteCarlo:


OpenCLGPU_MonteCarlo
ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_MonteCarlo.OpenCLAlteraFPGA_MonteCarlo

Public Member Functions

- def __init__
- def set_default_parameters (self)
- def generate_name (self)
- def generate_activity_thread (self)
- def generate_kernel (self)
- def compile
- def set_instance_paths (self, instance_paths)
- def set_chunk_paths (self)

Public Attributes

- simulation

    Boolean option for CPU simulation.
- optimisation

    Boolean option to use Altera OpenCL compiler optimisation flags.
- pipelining

    integer degree of loop unrolling to perform
- cslow

    boolean option for c-slowing
- instances

    integer number of instances
- simd_width

integer simd width to use
- **output_file_name**
- **random_number_generator**
- **instance_paths**
- **chunk_paths**

Static Public Attributes

- int instance_paths = 1

  Number of simulations to use per instance - analogous to the kernel path max used in OpenCL GPU class.

6.14.1 Detailed Description

```
Monte Carlo solver class for Altera OpenCL SDK FPGA Platforms
```

6.14.2 Constructor & Destructor Documentation

6.14.2.1 def ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_MonteCarlo.OpenCLAlteraFPGA↩_MonteCarlo.__init__ (*self*, *derivative*, *paths*, *platform*, *reduce_underlyings* = `True`, *kernel_path_max* = `1`, *random_number_generator* = `"taus_boxmuller"`, *floating_point_format* = `"float"`, *instances* = `None`, *pipelining* = `None`, *cslow* = `True`, *simulation* = `False`, *default_points* = `4096`, *optimisation* = `False`, *instance_paths* = `None`, *simd_width* = `None`)

```
Constructor

Parameters
    derivative, paths, platform, reduce_underlyings, kernel_path_max, random_number_generator, floating_point_
    pipelining - (int) number of iterations of inner, path kernel loop to unroll
    cslow - (bool) option for turning on c-slowing optimisation
    simulation - (bool) option to compile implementation for CPU simulation (compiles much faster)
    optimisation - (bool) option to turn on various mathematical optimisations
    instance_paths - (int) number of paths to use per instance
    simd_width - (int) vector width to use
```

6.14.3 Member Function Documentation

6.14.3.1 def ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_Monte↩Carlo.OpenCLAlteraFPGA_MonteCarlo.compile (*self*, *override* = `True`, *debug* = `False`)

```
Overriding the compile method as the Altera command line compiler must be used for their SDK

Parameters
    override, debug - same as in OpenCLGPU_MonteCarlo class
```

6.14.3.2 def ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_MonteCarlo.OpenCLAlteraFPGA↩_MonteCarlo.generate_activity_thread (*self*)

```
Similiar to other solver classes - overriding the generate activity thread method
```

6.14.3.3 def ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_MonteCarlo.OpenCLAlteraFPGA↩_MonteCarlo.generate_kernel (*self*)

```
Overriding kernel generation method.

In this case, the parent method from the OpenCL GPU class is called, but the output is then modified.
```

6.14.3.4 def ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_MonteCarlo.OpenCLAlteraFPGA↩_MonteCarlo.generate_name (*self*)

```
Overriding method for generating name
```

6.14.3.5 def ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_MonteCarlo.OpenCLAlteraFPGA↩_MonteCarlo.set_chunk_paths (*self*)

```
Helper method for setting the number of chunk paths to use
```

6.14.3.6 def ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_MonteCarlo.OpenCLAlteraFPGA↩_MonteCarlo.set_default_parameters (*self*)

```
Helper method for setting default FPGA parameter values
```

6.14.3.7 def ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA_MonteCarlo.OpenCLAlteraFPGA↩_MonteCarlo.set_instance_paths (*self, instance_paths*)

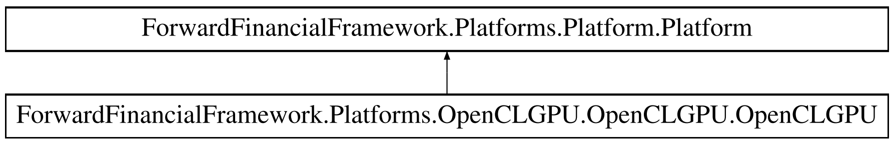
```
Helper method for setting number of instance paths

Parameters
    instance_paths - (int) number of instance paths to use
```

The documentation for this class was generated from the following file:

- OpenCLAlteraFPGA_MonteCarlo.py

6.15 ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU.OpenCLGP↩U Class Reference

Inheritance diagram for ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU.OpenCLGPU:


ForwardFinancialFramework.Platforms.Platform.Platform
ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU.OpenCLGPU

Public Member Functions

- def __init__

Public Attributes

- **threads**
- **platform_name**
- **platform**
- **device**
- **context**

Static Public Attributes

- string **name** = "opencl_gpu"
- int threads = 0

  The number of threads isn't used here, so the value is set to zero.
- device_type = pyopencl.device_type.ALL

  The OpenCL device type is set to ALL by default to pickup everything.

6.15.1 Detailed Description

```
OpenCL GPU Platform Class

The main utility of this class in ensuring the OpenCL device being targeted is actually present.
```

6.15.2 Constructor & Destructor Documentation

6.15.2.1 def ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU.OpenCLGPU.__init__ (*self*, *threads* = 0, *platform_directory_string* = "Platforms/**OpenCLGPU**/opencl_code", *root_directory_string* = None, *platform_name* = "", *device_type* = pyopencl.device_type.GPU, *ssh_alias* = "", *remote* = False, *hostname* = None)

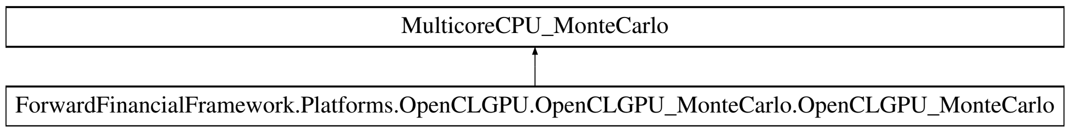
```
Constructor

Parameters
    platform_directory_string, root_directory_String, ssh_alias, remote, hostname - same as Platform class
    platform_name - (string) name of OpenCL SDK to use
    device_type - (pyopencl.device_type) OpenCL device type to use
```

The documentation for this class was generated from the following file:

- OpenCLGPU.py

6.16 ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU_MonteCarlo.↩ OpenCLGPU_MonteCarlo Class Reference

Inheritance diagram for ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU_MonteCarlo.OpenCL↩ GPU_MonteCarlo:


MulticoreCPU_MonteCarlo
ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU_MonteCarlo.OpenCLGPU_MonteCarlo

Public Member Functions

- def **__init__**
- def generate
- def generate_activity_thread (self)
- def generate_kernel (self)
- def generate_variable_declaration (self)
- def compile

Public Attributes

- **random_number_generator**
- **activity_thread_name**
- **kernel_code_string**
- **cpu_seed_kernel_code_string**
- **floating_point_format**
- **kernel_loops**
- **header_string**
- **kernel_code_list**
- **program**

6.16.1 Detailed Description

```
OpenCL GPU Monte Carlo Solver class

This class provides the generation, compilation and execution behaviours for OpenCL GPU platforms (including X
The Multicore solver class is reused heavily, with only the activity thread being implemented differently.
```

6.16.2 Member Function Documentation

6.16.2.1 def ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU_MonteCarlo.OpenCLGPU_MonteCarlo.compile (*self*, *override* = True, *cleanup* = True, *debug* = False)

```
Compiler method for OpenCL solver.

        In addition to compiling the host code, it compiles the OpenCL binary.

        Parameters
override, cleanup, debug - same as in Mutlicore CPU class
```

6.16.2.2 def ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU_MonteCarlo.OpenCLGPU_MonteCarlo.generate (*self*, *override* = True, *verbose* = False, *debug* = False)

```
Generate solver method

In addition to calling the Multicore CPU solver class to generate the host code, the kernel code is also gener

Parameters
    override, verbose, debug - same as in MulticoreCPU_MonteCarlo class
```

6.16.2.3 def ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU_MonteCarlo.OpenCLGPU_MonteCarlo.↩ generate_activity_thread (*self*)

```
Helper method for generating activity thread

        Overrides the method in MulticoreCPU_MonteCarlo
```

6.16.2.4 def ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU_MonteCarlo.OpenCLGPU_MonteCarlo.↩ generate_kernel (*self*)

```
Helper method for generating OpenCL kernel
```

6.16.2.5 def ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU_MonteCarlo.OpenCLGPU_MonteCarlo.←↩ generate_variable_declaration (*self*)

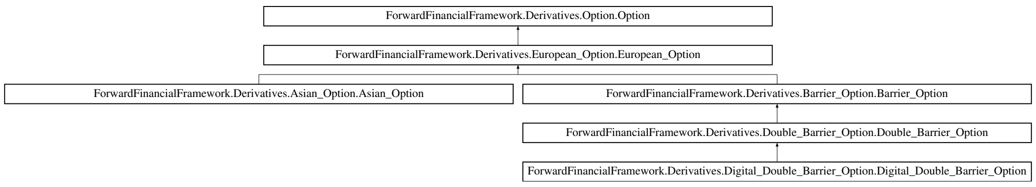
```
Overriding the helper method of the same name in the Multicore CPU solver class

Adding in struture for RNG
```

The documentation for this class was generated from the following file:

- OpenCLGPU_MonteCarlo.py

6.17 ForwardFinancialFramework.Derivatives.Option.Option Class Reference

Inheritance diagram for ForwardFinancialFramework.Derivatives.Option.Option:


ForwardFinancialFramework.Derivatives.Option.Option
ForwardFinancialFramework.Derivatives.European_Option.European_Option
ForwardFinancialFramework.Derivatives.Asian_Option.Asian_Option
ForwardFinancialFramework.Derivatives.Barrier_Option.Barrier_Option
ForwardFinancialFramework.Derivatives.Double_Barrier_Option.Double_Barrier_Option
ForwardFinancialFramework.Derivatives.Digital_Double_Barrier_Option.Digital_Double_Barrier_Option

Public Member Functions

- def __init__ (self, underlying, time_period, call, strike_price)
- def path_init (self)
- def path (self, price, time)
- def payoff (self, end_price)

Public Attributes

- **time_period**
- **strike_price**
- **value**
- **delta_time**

Static Public Attributes

- string **name** = "option"
- underlying = None

  The container for the Underlying upon which this product depends.
- float strike_price = 0.0

  The defined strike price of produce.
- float time_period = 0.0

  The time period of the product, i.e.
- call = None

  Call or put?
- float delta_time = 0.0

  The next time step required.
- float value = 0.0

  The value of the produce.

6.17.1 Detailed Description

```
Base derivative class

This class represents the base derivative class. In practice, its a simple European future.
```

6.17.2 Constructor & Destructor Documentation

6.17.2.1 def ForwardFinancialFramework.Derivatives.Option.Option.__init__ (*self*, *underlying*, *time_period*, *call*, *strike_price*)

```
Constructor

Parameters
    underlying - (list of FowardFinancialFramework.Underlyings) list of underlyings that this product depends
    time_period - (float) time until expiry of product
    call - (bool) call product?
    strike_price - (float) defined expiry price
```

6.17.3 Member Function Documentation

6.17.3.1 def ForwardFinancialFramework.Derivatives.Option.Option.path (*self*, *price*, *time*)

```
Path evolution method

        Parameters
price - (float) the current price of the underlying product
time - (time) the current time of the underlying product

        Evolves the derivative’s simulation. Is a dummy method for the base class
```

6.17.3.2 def ForwardFinancialFramework.Derivatives.Option.Option.path_init (*self*)

```
Path initialisation method

        Parameters
None

        Initiate the derivative’s path/simulation (resets value back to 0.0)
```

6.17.3.3 def ForwardFinancialFramework.Derivatives.Option.Option.payoff (*self*, *end_price*)

```
Payoff method

Parameters
    end_price - (float) the final price of the underlying

Finds the value of the product, based upon the end price of the underlying, and if this is a call or not.
```

6.17.4 Member Data Documentation

6.17.4.1 float ForwardFinancialFramework.Derivatives.Option.Option.time_period = 0.0 `[static]`

The time period of the product, i.e.

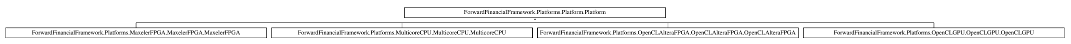
the time from the present until expiry

The documentation for this class was generated from the following file:

- Option.py

6.18 ForwardFinancialFramework.Platforms.Platform.Platform Class Reference

Inheritance diagram for ForwardFinancialFramework.Platforms.Platform.Platform:


ForwardFinancialFramework.Platforms.Platform.Platform
ForwardFinancialFramework.Platforms.MaxelerFPGA.MaxelerFPGA.MaxelerFPGA
ForwardFinancialFramework.Platforms.MulticoreCPU.MulticoreCPU.MulticoreCPU
ForwardFinancialFramework.Platforms.OpenCLAlteraFPGA.OpenCLAlteraFPGA.OpenCLAlteraFPGA
ForwardFinancialFramework.Platforms.OpenCLGPU.OpenCLGPU.OpenCLGPU

Public Member Functions

- def __init__
- def platform_directory (self)
- def root_directory (self)
- def absolute_platform_directory (self)

Public Attributes

- **platform_directory_string**
- **root_directory_string**
- **ssh_alias**
- **hostname**
- **shell_vars**
- **shell_setup_cmds**
- **shell_exit_cmds**

Static Public Attributes

- string **name** = "platform"
- string platform_directory_string = ""

  location of platform generated code
- string root_directory_string = ""

  root directory of F^3 on this system
- string ssh_alias = ""

  SSH alias for this system, i.e.
- remote = False

  Whether this is a remote executable or not.

6.18.1 Detailed Description

```
Base platform class

This class represents the base platform class. It contains all of the SSH specific communication commands.
```

6.18.2 Constructor & Destructor Documentation

6.18.2.1 def ForwardFinancialFramework.Platforms.Platform.Platform.__init__ (*self*, *platform_directory_string* = `None`, *root_directory_string* = `None`, *ssh_alias* = `""`, *remote* = `False`, *hostname* = `None`, *shell_vars* = `{}`, *shell_setup_cmds* = `[]`, *shell_exit_cmds* = `[]`)

```
Constructor

Parameters
    platform_directory_string - (string) location of platform specific code
    root_directory_string - (string) location of F^3 on this system
    ssh_alias - (string) SSH alias for this system, as stored in .ssh/sshconfig
    remote - (bool) Is this system remote?
    hostname - (string) - override the system hostname for generated code
    shell_vars - (dict) - set any environmental variables
    shell_setup_cmds (list of strings) - any commands that need to be run upon login
    shell_exit_cmds (list of strings) - any commands that need to be run upon logout

Upon construction, if this is a remote system, all of the SSH set up is done. It might take several seconds to
```

6.18.3 Member Function Documentation

6.18.3.1 def ForwardFinancialFramework.Platforms.Platform.Platform.absolute_platform_directory (*self*)

```
DEPRECATED return the path to generated code on this system
```

6.18.3.2 def ForwardFinancialFramework.Platforms.Platform.Platform.platform_directory (*self*)

```
DEPRECATED return the directory of the generated code
```

6.18.3.3 def ForwardFinancialFramework.Platforms.Platform.Platform.root_directory (*self*)

```
DEPRECATED return the directory of F^3 on this system
```

6.18.4 Member Data Documentation

6.18.4.1 string ForwardFinancialFramework.Platforms.Platform.Platform.ssh_alias = "" [static]

SSH alias for this system, i.e.

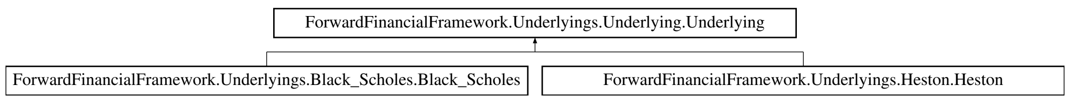
the entry in .ssh/sshconfig that F^3 will use to talk to this system

The documentation for this class was generated from the following file:

- Platform.py

6.19 ForwardFinancialFramework.Underlyings.Underlying.Underlying Class Reference

Inheritance diagram for ForwardFinancialFramework.Underlyings.Underlying.Underlying:


ForwardFinancialFramework.Underlyings.Underlying.Underlying
ForwardFinancialFramework.Underlyings.Black_Scholes.Black_Scholes
ForwardFinancialFramework.Underlyings.Heston.Heston

Public Member Functions

- def __init__ (self, rfir, current_price)
- def **__repr__** (self)
- def path (self, delta_time)

Public Attributes

- **rfir**
- **current_price**
- **gamma**
- **time**

Static Public Attributes

- string **name** = "underlying"
- float rfir = 0.0

  The Risk Free Interest Rate.
- float current_price = 0.0

  The current price.
- float gamma = 0.0

  Underlying log space state variable.
- float time = 0.0

  The current time distance from present of the underlying.

6.19.1 Detailed Description

```
Base underlying class

This class represents the base underlying type. In practice it simulates the behaviour of a simple, compound i
```

6.19.2 Constructor & Destructor Documentation

6.19.2.1 def ForwardFinancialFramework.Underlyings.Underlying.Underlying.__init__ (*self, rfir, current_price*)

```
Constructor

Parameters
    rfir - (float) the Risk Free Interest Rate
    current_price - (float) the starting price of the underlying
```

6.19.3 Member Function Documentation

6.19.3.1 def ForwardFinancialFramework.Underlyings.Underlying.Underlying.path (*self, delta_time*)

```
Path evolution method

Parameters
    delta_time - (float) the time step by which the price should be evoloved.
```

The documentation for this class was generated from the following file:

- Underlying.py