Autotuning Wavefront Patterns for Heterogeneous Architectures

Siddharth Mohanty

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2015

Abstract

Manual tuning of applications for heterogeneous parallel systems is tedious and complex. Optimizations are often not portable, and the whole process must be repeated when moving to a new system, or sometimes even to a different problem size.

Pattern based parallel programming models were originally designed to provide programmers with an abstract layer, hiding tedious parallel boilerplate code, and allowing a focus on only application specific issues. However, the constrained algorithmic model associated with each pattern also enables the creation of pattern-specific optimization strategies. These can capture more complex variations than would be accessible by analysis of equivalent unstructured source code. These variations create complex optimization spaces. Machine learning offers well established techniques for exploring such spaces.

In this thesis we use machine learning to create autotuning strategies for heterogeneous parallel implementations of applications which follow the wavefront pattern. In a wavefront, computation starts from one corner of the problem grid and proceeds diagonally like a wave to the opposite corner in either two or three dimensions. Our framework partitions and optimizes the work created by these applications across systems comprising multicore CPUs and multiple GPU accelerators. The tuning opportunities for a wavefront include controlling the amount of computation to be offloaded onto GPU accelerators, choosing the number of CPU and GPU threads to process tasks, tiling for both CPU and GPU memory structures, and trading redundant halo computation against communication for multiple GPUs.

Our exhaustive search of the problem space shows that these parameters are very sensitive to the combination of architecture, wavefront instance and problem size. We design and investigate a family of autotuning strategies, targeting single and multiple CPU + GPU systems, and both two and three dimensional wavefront instances. These yield an average of 87% of the performance found by offline exhaustive search, with up to 99% in some cases.

Lay Summary

Modern computing systems are increasingly becoming hybrid systems consisting of multiple CPU cores and specialized accelerators such as graphical processing units (GPU) that can offload many intensive operations to offer higher processing capability than a single core system. The presence of many CPU and GPU cores allows a large chunk of work to be partitioned into many smaller chunks that can be computed simultaneously or in parallel. Parallel computing can be seen in action in almost every field that requires large scale computation to be carried out as quickly as possible. Its application ranges from handling millions of transactions concurrently in an on line trading platform to making accurate weather forecasts that require massive computation power. IBM Deep Blue and Google data centers are all examples of such systems.

However, programming such hybrid systems is complex. This is because the programmer now has to decompose serial problems into parts that can be computed in parallel. After conceptually decomposing the problem, the programmer is then required to become an expert in multiple, conceptually diverse languages and libraries to implement parallelism, and to integrate these within single applications. An application programmer would prefer not to be bogged down by the low level details of problem distribution across systems and rather focus on the computation. And, even after implementing parallelism, the code needs to run optimally across different types of hybrid architectures so that the programmer does not have to expend time on tuning the application for each architecture.

Our work addresses these two issues of implementing parallelism transparently and making code performance portable across diverse heterogeneous systems. We provide a framework that lets the programmer write application level code while our framework handles the distribution of work across multiple CPU cores and GPU devices. We target a specific class of problems called wavefronts which can be difficult to parallelise due to their internal dependencies that place restrictions on how the problems can be decomposed and distributed. After implementing parallelism, our framework automatically selects high performing configurations for the application so that the programmer does not have to manually port each application to a new system. Our autotuning framework shows promising results based on numerous trials across three different classes of applications and five diverse systems.

Acknowledgements

Much credit for the research undertaken is due to Dr. Murray Cole who guided and provided valuable inputs to this project over numerous mails and meetings. This research undertaking was possible due to generous grants from the Institute for Computing Systems Architecture (ICSA), Numerical Algorithms and Intelligent Software Centre (NAIS), Ethel Gordon Fund, the University of Edinburgh College Science Fund and the Chalmers Fund. Finally, I would like to thank my family for their unconditional support, my friends for their encouragement and camaraderie and Shishu for always being there for me.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published earlier as follows:

- Siddharth Mohanty and Murray Cole. Autotuning wavefront applications for multicore multi-GPU hybrid architectures. In Proceedings of Programming Models and Applications on Multicores and Manycores, PMAM'14, pages 1:1-1:9, New York, NY, USA, 2014. ACM [77].

- Siddharth Mohanty and Murray Cole. Autotuning wavefront abstractions for heterogeneous architectures. In Applications for Multi-Core Architectures (WAMCA), 2012 Third Workshop on, pages 42-47. IEEE, 2012 [76].

- Siddharth Mohanty. Task farm optimization through machine learning. Diss. Master's thesis, Institute of Computing Systems Architecture, Informatics Department, University of Edinburgh, 2010 [75]. Paraphrased materials from the thesis with additional content have been used in chapter 2 and chapter 7. Specifically, these are subsection 2.2.1, subsection 2.2.2, section 2.3, subsection 2.4.2, subsection 2.4.3, subsection 2.4.4, subsection 2.4.5, subsection 2.4.6, section 2.7, subsection 7.2.3 and the last paragraph of section 7.4.

Siddharth Mohanty

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel computing literally means carrying out computations in parallel, i.e at the same time and encompasses the hardware aspect of dealing with the type and number of processors, and the software side of solving problems through computational models like shared memory or message passing [71].

Parallel computing can be seen in action in almost every field that requires large scale computation to be carried out as quickly as possible. Its application ranges from handling millions of transactions concurrently in an on line trading platform to making accurate weather forecasts that require massive computation power. IBM Deep Blue and Google data centers are all examples of such systems.

One of the earliest definitions of what constituted a parallel processor was provided by Almasi [2] who defined it as *"a collection of serial processors which can communicate and cooperate to solve big problems fast".* This old world high performance computing (HPC) landscape that envisioned large clusters of single core processors communicating over networks to distribute work and compute in parallel has been extended by heterogeneous architectures consisting of multicore CPUs and diverse accelerators that have made parallelism mainstream [6].

Thus the modern parallel landscape can be thought to be composed of three common forms of parallelism which typically overlap and can be merged in various combinations in real heterogeneous systems. The first form of parallelism is obtained from integration at the chip level such as the chip-multiprocessing as seen on Sun's UltraSPARC T2 Plus 8-core processor [35]. The next form of multiprocessing capability is obtained through clusters of workstations, for example the Beowulf Cluster [96] or the Edinburgh Compute and Data Facility (ECDF) [8] - a high-performance cluster of multicore nodes with 1456 processors that runs a Batch System where a user submits a job and waits in a queue until the resources it requires are available. Other examples of such multicore cluster systems are the CRAY XE6 series of supercomputers [41]. The third form of parallelism is seen in specialized accelerator devices including GPGPUs and FPGAs. A three tier form of parallelism (multicore-cluster-accelerator) can be seen in the system shown in 1.1. The supercomputing facility Tianhe [9] and the CRAY CS series of supercomputers [40] are examples of such clusters of accelerator workstations. There are more numerous examples of heterogeneous systems without the cluster level, such as modern personal computers and Ultrabook laptops that typically comprise of multicores and multiple GPUs. Our experimental systems also had two overlapping forms of parallelism - chip level and accelerator level.

Among accelerators, GPUs are increasingly ubiquitous as they have evolved from being used purely for graphical rendering in computer gaming to being used for non graphical processing for HPC applications. Taking advantage of the parallelism afforded by their hundreds of cores, GPUs have become cost effective alternatives to HPC hardware for applications whose problem structure is inherently data-parallel. Examples of such HPC applications include quantum chromo-dynamics experiments in physics [10] , gene-sequencing in bio-informatics [3], competitive game theory problems in financial strategy [104] and incompressible flow computations [97]. However, using these specialized accelerators along with multicores brings its own challenges, as discussed in the next section.

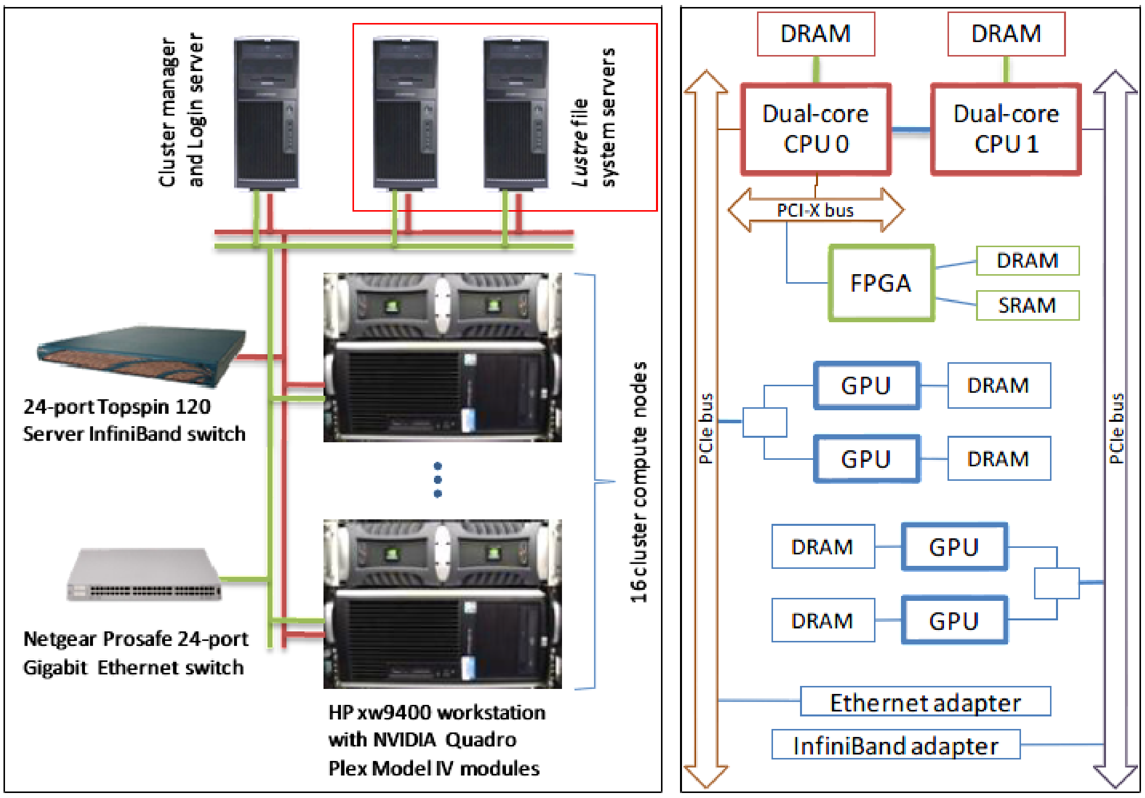

Cluster manager and Login server
Lustre file system servers
24-port Topspin 120 Server InfiniBand switch
Netgear Prosafe 24-port Gigabit Ethernet switch
16 cluster compute nodes
HP xw9400 workstation with NVIDIA Quadro Plex Model IV modules
DRAM
DRAM
Dual-core CPU 0
Dual-core CPU 1
PCI-X bus
FPGA
DRAM
SRAM
GPU
DRAM
GPU
DRAM
DRAM
GPU
DRAM
GPU
PCIe bus
PCIe bus
Ethernet adapter
InfiniBand adapter

Figure 1.1: Heterogeneous Computing Architecture, taken from [52]

1.1 Heterogeneous Parallelism Challenges

The Heterogeneous Computing paradigm offers challenges in the form of the twin issues of problem decomposition-distribution and performance portability.

- **Problem Decomposition and Distribution** : MultiCore accelerator architectures pose problems to the application developer and programming and systems tool-chains. Firstly, the developer has to be able to decompose serial problems into parts that can be computed in parallel. After conceptually decomposing the problem, the programmer is then required to become an expert in multiple, conceptually diverse languages and libraries to implement parallelism, and to integrate these within single applications.

  To exploit parallelism, an application programmer would prefer not to be bogged down by the low level details of handling communication and cooperation between processes across systems (problem distribution). Instead the primary focus of the programmer should be the computation task at hand. These twin issues can be addressed by an API or application programming interface. Besides getting ease of use from this layer of abstraction, the application programmer would also require this interface to be efficient which means, optimally exploiting all the available resources in the system. This brings up the next issue.

- **Performance Portability** : Performance tuning of such applications is complex, and typically much more architecture specific than in simpler, essentially homogeneous systems. The layer of abstraction that handles problem decomposition and distribution should ideally perform optimally across heterogeneous systems where the three forms of parallelism overlap and this performance portability should be transparent to the end user who should not be tasked with porting an application for each architecture.

Finding a programming methodology and toolchain which can address these challenges is widely recognized as being of major importance, both academically and industrially [14].

1.2 Contributions of this Thesis

Pattern-oriented parallel programming has experienced a recent upsurge in interest [72], and offers a promising approach to the twin challenges of heterogeneous parallelism mentioned earlier. Each pattern of interest is abstracted and encapsulated behind an API which requires the programmer to code only application specific aspects. This approach simplifies the programmers task (there is no parallelization “glue” code to write), and presents the system with a constrained optimization challenge of choosing between and tuning internal parameters of a set of pre-existing candidate, heterogeneous parallelizations.

We present a case study in the application of this approach by employing patterns in the form of algorithmic skeletons [20]. We chose skeletons because they are flexible and reusable constructs (skeletons can be nested inside other types of skeletons to create new patterns and they can be highly abstract or detailed). Skeletons handle the communication-synchronization mechanism of concurrent operations, thus providing transparent problem decomposition and distribution. By being architecture-specifically tunable they also allow performance portability. We have provided background details of skeletons and the advantages of employing skeletons in section 2.3.

Our selected pattern is the *wavefront*, which captures a specific class of dynamic programming problems. We are interested in this pattern because the dependencies inherent in a wavefront make it challenging to parallelize across hybrid settings of multicore CPU processors and GPU accelerators (as compared, for example to embarrassingly parallel stencil patterns). A large number of applications exhibit the wavefront pattern across diverse domains including bio-informatics, financial game theory, particle physics and linear algebra. However, there has as yet been no extensive investigation into the tuning of wavefront patterns across heterogeneous systems. This means tuning wavefronts in such hybrid systems is an important area of research. The relevant background details regarding wavefront dependencies and applications are introduced in detail in subsection 2.5.1.

Our implementation strategy distributes wavefront applications across systems which incorporate a multicore CPU and multiple GPU accelerators. In order to better understand the tuning trade-offs, and to assist in the evaluation of our heuristics, we have performed an exhaustive exploration of an interesting fragment of the tuning space, across a collection of systems comprising a CPU and single or multiple GPUs. Since such an exhaustive search would be impractical in a production system, we have investigated the application of machine-learning strategies to reduce the search time. We have experimented across a range of wavefront applications and heterogeneous systems.

There are two main contributions of this thesis.

Designing tunable high level parallel programming models for wavefront applications

In many high level parallel programming models employing skeletons, there are several opportunities for optimization. In one study of Intel Threading Building Blocks (TBB) [22], it was reported that increasing the number of cores increases the overhead costs to become as high as 47 percent of the total per-core execution time on a 32-core system. This highlights the need to tune the parameters of these models in order to avoid performance penalties. In this thesis we have identified what parameters can affect the runtime of our wavefront skeleton, and how to tune them.

Demonstrating effectiveness of machine learning for optimizing wavefront ab-

stractions on heterogeneous architecture
Many automated optimizations based on prior learning have been done at the low level for applications that are dependent on the hardware. However there are few that target high level parallel programming models. Utilization of these techniques in exploring the parameter space of wavefronts for heterogeneous architectures has provided empirical data showing how effective machine learning is for tuning a high level model like the wavefront.

1.3 Structure of the Dissertation

The remainder of the thesis is structured as follows

- Chapter 2 provides all the background details related to the need for algorithmic skeletons to solve the parallel programming crisis, tuning skeletons through machine learning to enable performance portability, dynamic programming problems and terminologies associated with a special case of dynamic programming problem, the wavefront pattern and its applications.

- Chapter 3 discusses our skeleton or framework implementation strategy for 2D wavefront application in multicore CPU + single GPU and multicore CPU + multiple GPU systems. It then provides the details of implementing a 3D wavefront application in multicore CPU + single GPU systems. It also discusses our experimental setup, performance trade-offs in our framework and how they are tuned by using machine learning techniques.

- Chapter 4 first delves into the tuning strategy of a 2D wavefront application in a multicore CPU + single GPU environment. It then discusses the results of our exhaustive search of the tuning space and the evaluation of machine learning strategies for autotuning.

- Chapter 5 similarly delves into the experimental details of autotuning 2D wavefront applications in a hybrid multicore CPU + multiple GPU environment. The addition of another GPU increases the complexity of tuning with more tunable parameters and decisions to be taken.

- Chapter 6 provides autotuning experience for 3D wavefront applications. The results of an exhaustive search of optimal performing points are analyzed, and machine learning results for our autotuner are presented.

- Chapter 7 discusses related work, comparing our autotuning methodology and results with other similar autotuners.

- Chapter 8 concludes the thesis with a review of our contributions and discusses future work.

Chapter 2

Background

This chapter provides background information on topics relevant to our work. These relate to the opportunities and challenges of parallelism, discussed in section 2.1, followed by details of common parallel programming paradigms and their drawbacks in section 2.2. The limitations of commonly used parallel models highlight the need for abstractions. We introduce algorithmic skeletons, their classification, their guiding design principles, and address the issue of performance portability of skeletons in section 2.3. In section 2.4 we explain the need for predictive modeling based on the curve fitting concept, followed by an overview of some machine learning models that have been employed in this thesis. Since the wavefront pattern is a specific class of dynamic programming problem, we provide the necessary background to dynamic programming problems in section 2.5. This section then provides the conceptual details of the wavefront abstraction that forms the core of our thesis. The corresponding wavefront applications used in our work are described in section 2.6. Finally we conclude this chapter with a discussion of the measurement of parallel processing performance in section 2.7.

2.1 Parallel Computer Architecture

2.1.1 Opportunities and Challenges of Parallelism

From late 20th century (1970 onwards), CPU processing capability has been steadily improving at exponential rates and is expected to continue until 2020 [85]. This observation is referred to as Moore's law which states transistor counts in integrated circuits double approximately every two years. The resulting increase in clock rates enables single-threaded code to execute faster on every new generation of processors with no modification. However, in 2005 Moore noted that transistors would eventually reach the limits of miniaturization at atomic level [86], i.e. transistor scaling would saturate owing to limited gate metals, limited options for channel material and limited material solutions for reducing source to drain leakage. Meanwhile, new computing prototypes such as quantum computing are at experimental stages and are a long way from becoming mainstream [38].

There is also the economic cost to increasing miniaturization. Rock's law states the capital cost of setting up a semiconductor fabrication plant increases exponentially with every new generation of chips [90]. Another constraint to Moore's law of adding more transistors to conventional CMOS chip is related to energy and power dissipation [30]. This is because increased transistor density leads to increased internal processor heat that results in overheating and excess energy use. Recent work in addressing this includes [30], that explores a design space of matching supply voltage with threshold voltage of the transistor to enable energy saving. This design however suffers from performance loss, as measured by the fanout-of-four inverter delay or F04 metric, which is 10 times slower compared to conventional supply voltage design.

It should also be noted that merely doubling CPU capacity does not automatically improve performance. Memory bandwidth also plays a crucial role based on Von Neumann's limit [81]. So sequential computational performance is subject to the available memory bandwidth. The graph below underscores the increasingly expensive sequential memory access cost.

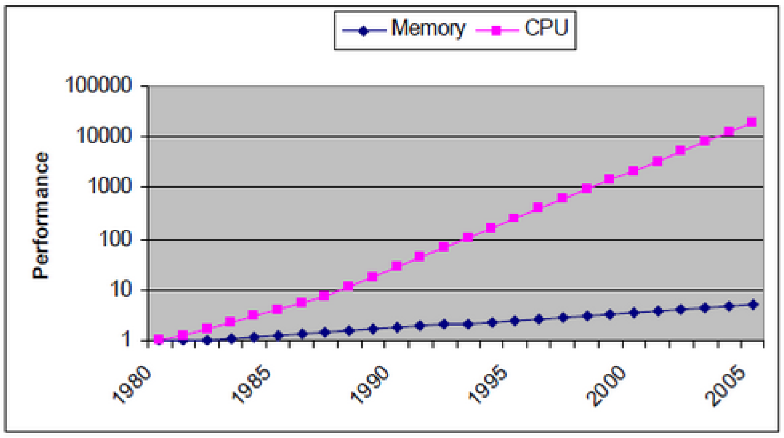


Figure 2.1: The Memory Gap, taken from [52]

One solution to the costly process of manufacturing faster chips that are energy efficient design and have improved memory bandwidth is to employ parallelism. The overall time taken to compute a large task can be quicker with multiple cheaper, but relatively slower, processors working on parts of the problem rather than one fast expensive processor computing the whole task. Based on our previous discussion of three tiers of parallelism in chapter 1, we discuss how each tier of parallelism can outperform sequential processing and also compare their relative strengths and weaknesses.

2.1.1.1 Multiple Node Architecture

In the traditional HPC parallelism, clusters of nodes form a distributed address space. The distribution of large tasks and chunks of data across nodes reduces the memory load on individual nodes and overcomes sequential bottlenecks. This performance is limited by the network speed but with high speed connections (Infiniband/Ethernet) the traditional HPC based parallelism can outperform sequential processing, subject to Amdahl's law.

2.1.1.2 Shared Memory Architecture

Next we look at shared memory systems consisting of a single node but multiple processing cores that share a single address space. The immediate benefit over a sequential single core system is the greater computing capacity afforded by the multiple cores present on a chip. Compared to traditional HPC clusters, they also do not suffer from network bottlenecks of distributed systems. However, shared memory systems incur the additional architectural overhead of maintaining cache coherency and the programmers are also burdened with understanding the intricacies of relaxed consistency memory models needed for cache coherent (CC) systems. The time and memory cost of scaling the number of cores on a CC system quickly grows beyond a point at which additional cores are not useful in a single parallel problem. This is referred to as the *Coherency Wall* [70].

2.1.1.3 Manycore Architecture

We now look at many core computing or massively multi-core systems which refer to systems having processors numbering hundreds to thousands, with the exception of the traditional distributed systems. These include shared memory systems with high processor counts and accelerator based (GPU, FPGA) systems.

An example of such a shared memory many core system is the 48 core Intel Single Chip Cloud platform [80] which internally implements message passing between its cores and can be considered to possess a hybrid address space.

Accelerator based heterogeneous architectures consist of one or more GPGPUs or FPGAs. Due to the availability of hundreds of compute units on GPUs, these systems can outperform sequential processing. They also do not suffer from network bottlenecks of distributed systems but global memory contention and data transfer overheads. These systems are also limited to SIMD parallelism (discussed later in subsection 2.2.3) and are constrained by bandwidth limitations of PCI-E. Improvements to PCI-E layout in [68] helps reduce this bandwidth overhead. [16] demonstrates heterogeneous GPGPU parallelism wins over sequential memory access bottlenecks by adding memory layout remapping that takes advantage of concurrent GPU operations while overlapping with PCI-E memory transfer.

2.2 Parallel Programming Paradigms

The benefit of parallelism is obtained after problem decomposition has been carried out. This is subject to Amdahl's constraint of parallelism being limited by the sequential parts of the code [53]. Both problem decomposition and the subsequent work load distribution are challenging tasks, but the performance gains with a parallel system are often many times higher than the single core system. Keeping the need for parallelism in mind we now delve into the many conceptually distinct programming models that address parallelism, to illustrate how challenging it is to integrate or port to these very different models. We address those differences by first discussing the two leading programming paradigms for systems based on clusters and multicores - Message Passing (MP) and Shared Address Space (SAS) [92]. Then we discuss the accelerator programming languages for GPUs - CUDA and OpenCL.

2.2.1 Message Passing Paradigm

We begin with the message passing model which gives each process its own address space. The task is to distribute data across these address spaces and provide communication between processes by sending and receiving messages.

In the world of Message Passing, MPI has been enjoying the position of de facto standard since its inception [51]. Even recent programming systems like MPJ Express that target multi-core processors are, nevertheless MPI hybrids having MPI-like libraries [91]. As a distributed memory programming model, MPI is based on explicit control parallelism as defined by the MPI standard [15], making it a language independent communications protocol. It supports point to point communication that involves two processes in a process group and collective communications that involve all processes in a process pool. MPI subroutines can be called from any language that is able to interface with its libraries since MPI uses language independent specifications for calls. Hence there are C/C++, Fortran, Java and Python implementations of MPI.

A sample point to point MPI operation for sending data is as follows : MPI _Send (start, count, datatype, destination, tag, communicator) in which the message buffer has a starting address, the number of elements in the count and the size of each element as per the datatype. The target process rank (a number between 0 and MPI _COMM _SIZE -1 where the latter is the total number of processes in the global MPI communicator) is given by destination and tag can be used to obtain status information or send any additional data.

Figure 2.2 shows both point to point communication in the form of MPI _Send, MPI _Receive and the collective operation MPI Broadcast among three nodes in a cluster. MPI _Send enables a process to send a message to another process which receives the sent message through MPI _Receive. The Send-Receive operations can be blocking or asynchronous non

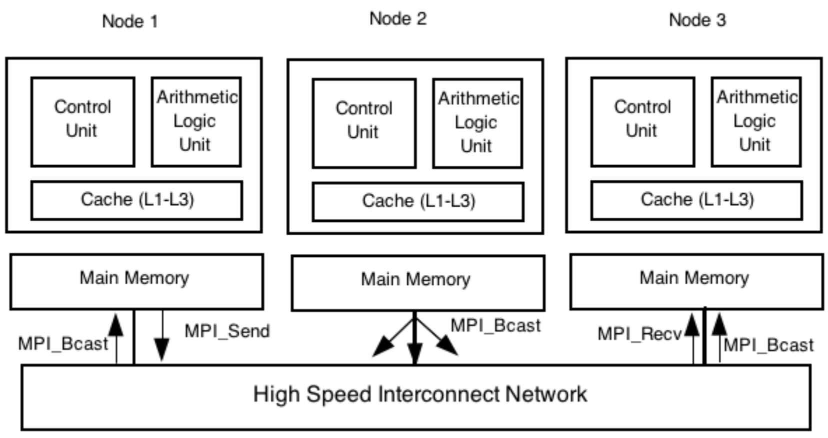


Figure 2.2: The MPI model : 3 nodes communicating over a high speed interconnect using point to point communication like MPI_Send from Node 1, MPI_Recv to Node 3, and the MPI_BCAST collective communication from Node 2 to Nodes 1 and 3.

blocking. In MPI collective communications like MPI_BCAST, every process including the receiver processes of Node 1 and Node 3, have to participate by calling the broadcast routine which has a parameter specifying the *root* or the sender process (Node 2 in the figure).

MPI _Bcast call takes data from one node and sends it across all other nodes in the process group. The data transferred during these operations can be predefined MPI datatypes like MPI _INT, MPI_CHAR or custom defined ones. A key feature of MPI is its dynamic process management, which establishes connection between MPI processes that have been independently spawned. MPI is thus well suited to task parallel applications such as the Task Farm where worker nodes can independently start working on chunks of work and return results to a master node.

2.2.2 Shared Address Space (SAS) Paradigm

The shared address space model allows threads or lightweight processes to directly interact through shared memory locations. This model raises the issue of memory consistency i.e., when and in what order should updates to memory made by one processor become visible to other processors. There are also race conditions to be dealt with, where threads compete against each other for access to the shared resource.

C's POSIX threads or Pthreads [11] form the standard for thread programming in which the main thread/process can start, synchronize and stop other threads of activity within its address space. Pthreads is a standardized model for interleaving the execution of sub-tasks partitioned from a main task. Parallelism is achieved as each thread begins the execution of a given function, which either terminates upon that function's exit or is terminated by another thread. A sample Pthread flow is shown in figure 2.3 where the main thread creates three different threads to work on three sub tasks using pthread _create. This method is of the form

$$pthread\_create(thread, attr, start\_routine, arg) \tag{2.1}$$

thread refers to a unique identifier for new threads. *attr* deals with various properties like scheduling policy, parameters and contention, stack size, address etc. The *start_routine* is basically a C function pointer. *arg* deals with arguments to the subroutine, passed by reference as pointer cast of type void. To synchronize the execution of the spawned threads, the main thread calls pthread_join , which suspends the main caller thread until the spawned thread exits. Once all three subtasks are completed, the caller thread resumes execution. Since all three threads access the same memory location, entry to the critical section is restricted to one thread at a time through mutexes that enforce mutual exclusion with mutex lock and unlocks.

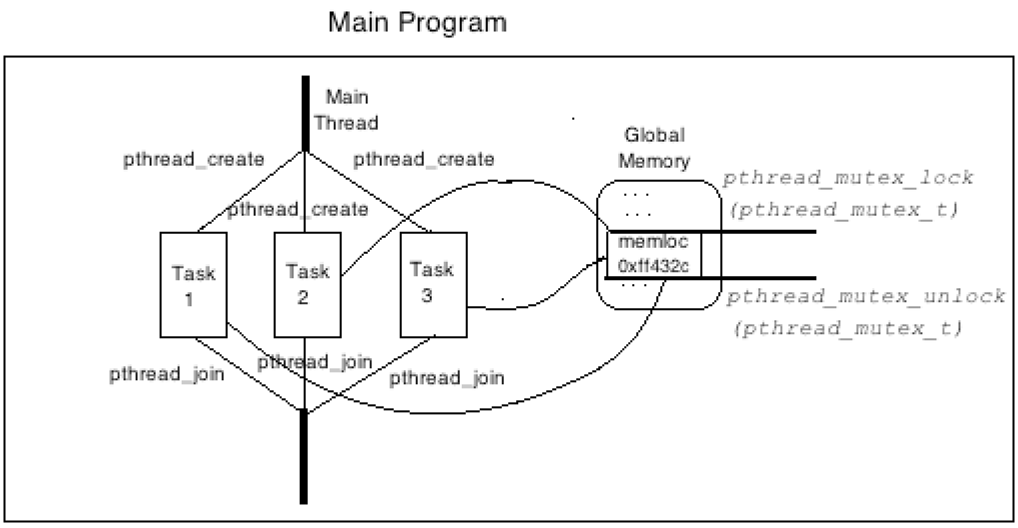


Figure 2.3: Three threads being spawned through pthread _create to work on 3 subtasks independently, using a mutex to safely access the shared memory location and update shared variables, before being joined to the main spawning thread using the pthread _join synchronization mechanism

The Pthreads standard specifies concurrency, meaning tasks defined by the programmer can occur before or after another or in parallel, depending on the operating system and the hardware they run on. Thus Pthread programs can run on both single cores and multicores.

The OpenMP [24] application program interface (API) is popular for the SAS paradigm, particularly for multi cores. It is a portable programming interface in C/C++ or Fortran and supports multi-platform parallel programming on all architectures.

OpenMP implements multi-threading by spawning a specified number of worker threads from a master thread and divides tasks among them. The threads run concurrently and are allocated to different processors by the runtime environment. This is illustrated in Figure 2.4 In OpenMP threads are created using the pragma directive **omp parallel**. OpenMP provides

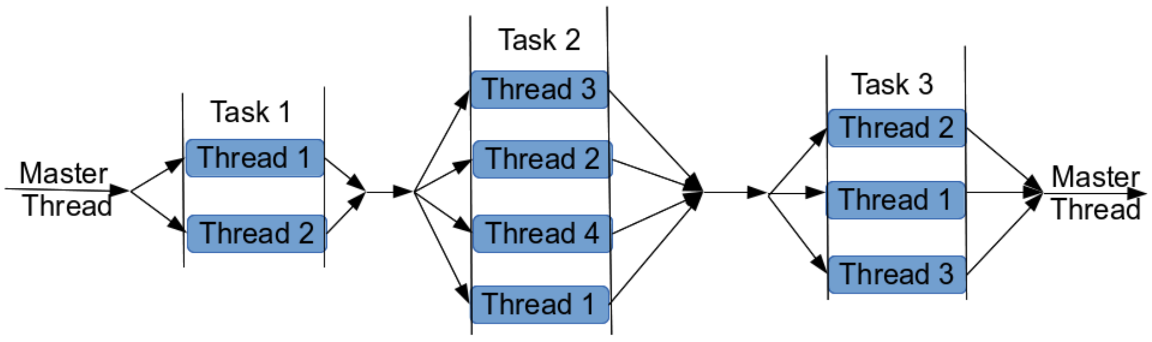


Figure 2.4: Master thread forking worker threads using pragma omp parallel directive and tasks assigned to threads, which are themselves bound to different processors by the runtime. These worker threads synchronize into the master thread without the need for explicit join statments as in pthreads.

work-sharing loop constructs like *omp for* and *omp do* that partition loop iterations among threads. Shown below is a code listing that demonstrates loop splitting among 10 threads using parallel for. Each element in an array of size 10 creates a task which is assigned to implicit OpenMP threads that independently squares that element.

```
// example of loop construct : omp parallel for
#include <omp.h>

void main(int argc, char *argv[]) {
    const int NUM_THREADS = 10;
    int i, demo[NUM_THREADS];

    #pragma omp parallel for
    for (i = 0; i < NUM_THREADS; i++)
        demo[i] = i * i;

}
```

Unlike Pthreads, here the programmer does not have to worry about explicitly creating and synchronizing the spawned threads. However, OpenMP provides synchronization clauses like static scheduling where threads are allocated loop iterations before executing or dynamic scheduling where smaller number of threads dynamically fetch new iterations after completing their initial allocated iterations. OpenMP also provides data sharing attribute clauses, such as shared data region which is accessible by all threads or private data region where each thread have their own local copies of variables.

Intel's Threading Building Blocks (TBB) [31] is also becoming a popular alternative. It provides easy to use thread safe container classes and templates that abstract out low level details such as multiple thread synchronization and load balancing among processes. Unlike the low level threading API Pthreads, the TBB library provides high level abstractions in the form of algorithms. Like OpenMP, TBB provides loop templates such as *parallel_for, parallel_reduce, parallel_do* etc. These loop templates are powered by its runtime engine called the *task scheduler*. TBB operations are treated as tasks that are allocated to multi-core processors dynamically by the library's run time engine which efficiently utilizes the CPU cache. Thus, TBB employs logical tasks instead of the usual logical threads that map onto the physical threads of hardware, which are lightweight compared to logical threads. The time taken to spawn and terminate a task has been measured at 18 times faster than the thread creation and termination in Unix systems, and on Windows system the ratio exceeds 100 [69]. The reason for this is logical threads require a local copy of register state, stack and a process identifier (in Linux). TBB tasks in contrast, are small routines which can not be preempted at task level. Besides, thread schedulers typically distribute time slices in a fair round robin fashion as this is a safe strategy that does not require high level understanding of the program structure. Since TBB provides abstractions mapped to algorithms, the tasks have some higher level information which allows the task scheduler to employ a greedy work-stealing algorithm resulting in higher efficiency. The tasks are executed respecting their internal graph dependencies and the scheduler assigns tasks to the underlying threading API (Pthreads in Posix systems), providing efficient load balancing. TBB also provides atomic operations and built in atomic template classes that provide better performance compared to manually enforcing atomicity in OpenMP [28]. A sample parallel_for is shown in the below code listing.

```
#include "tbb/tbb.h"

void ParallelApplyFoo( float a[], size_t n ) {
   parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a));
}
```

The entire iteration space 0 to n-1 represented by the blocked _range construct is divided into subspaces for each processor by the parallel _for loop template. ApplyFoo is the functor (whose implementation is not shown for sake of brevity), that is applied to each resulting sub-range of the array of floats 'a'. With this discussion we conclude the subsection on shared address spaces and move on to the accelerator programming paradigm.

2.2.3 Accelerator Programming Paradigm

While scaling complex CPU cores to hundreds or thousands is hard, there are already thousands of simpler RISC based cores working in a SIMD fashion [61] inside GPUs. Early GPUs were used for graphic processing activities like rendering images, but with heterogeneous computing becoming mainstream, general purpose GPU computing has increased in usage. However, due to the single instruction multiple thread limitation of GPU architectures where a single instruction executes on multiple work-items, having any kind of control flow like $if-else$ blocks means all work-items execute the if block first and then all of them execute the $else$ block. This contrasts with pthreads in CPU multi-cores that operate in SPMD (single program multiple data) fashion. A pthread can be assigned to the if block and another to else block and both will execute simultaneously.

Software implementations for programming GPUs are based on either the CUDA programming language, which is specific to NVIDIA graphic cards or OpenCl programming language [34] which is platform agnostic. Another implementation of heterogeneous computing is the Asymmetric Distributed Shared Memory (ADSM) programming model that “maintains a shared logical memory space for CPUs to access objects in the accelerator physical memory but not vice versa” [45]. This allows for light weight implementation, reduced programming effort and increased portability.

OpenCL and CUDA are the two dominant GPU programming languages and our auto tuning framework is built on OpenCL as it is more portable compared to CUDA. It should be noted that CUDA and OpenCL programming models are SIMT or Single Instruction Multiple Thread which is more flexible than SIMD with some additional costs. SIMT has scalar syntax instead of vector syntax, so the code is written for a single thread using standard arithmetic operators, instead of assembly-like opcodes in SIMD loops.

An example of an OpenCL kernel or computation function which is executed inside the GPU is shown below.

```
__kernel void constant_add_gpu (const int num,
           __global const float* src_a,
                 __global float* res
        )
{
   /* get_global_id(0) returns the ID of the thread in execution */
   const int id = get_global_id(0);

   /* Now each work-item performs the corresponding computation of adding num to
       itself */
      res[id] = src_a[id] + num;
}
```

SIMT allows a single instruction to be applied to multiple addresses enabling indirect memory access in the form of $a[b[i]]$ where the index for array a is computed from $b[i]$. Finally SIMT allows flow divergence. This is explained in the code listing below.

```
__kernel void constant_apply_even (const int len,
        __global const float* src_vector,
            __global float* res
      )
{
  /* get_global_id(0) returns the ID of the thread in execution */
  const int id = get_global_id(0);

  /* Now only even work items will compute */
  if(id%2==0) //Flow Divergence
    res[id] = src_vector[id] + id
}
```

In a GPU, the work item is the smallest entity of execution (analogous to a thread) which has an ID to distinguish the data being computed by it. The programmer specifies the number of work items to be spawned upon launching the kernel and each work item executes the same piece of kernel code. Work items can be synchronized inside work-groups that allow the work items to communicate and cooperate as shown in figure 2.5.

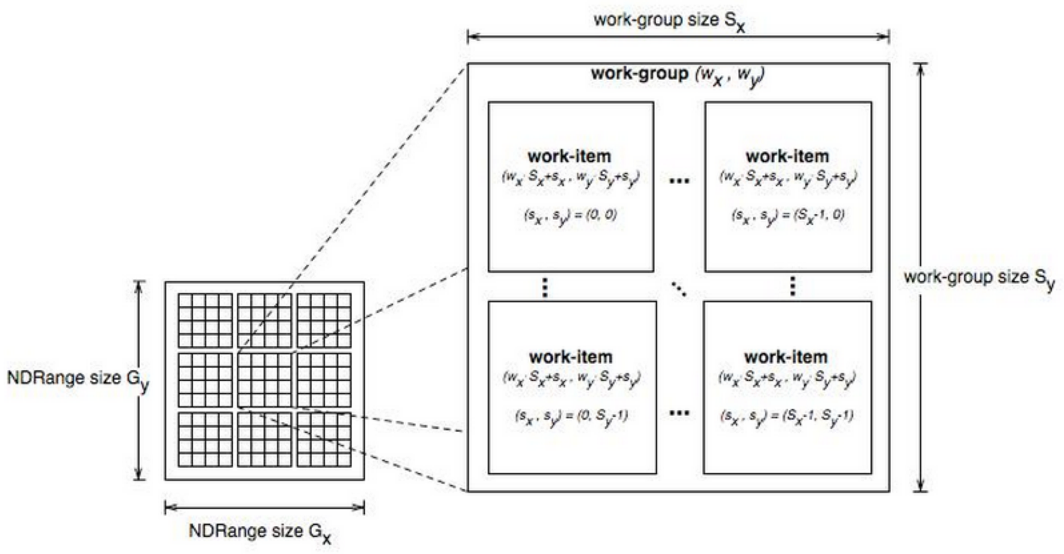

work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x S_x+s_x, w_y S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x S_x+s_x, w_y S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x S_x+s_x, w_y S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x S_x+s_x, w_y S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

Figure 2.5: Spawned work items organized into a 2D grid of work groups. The number of work groups are specified by the NDRangeKernel which enqueues a command to execute a kernel on a GPU or even CPU device, adopted from [44]

The programmer has to manage transferring data to and from the GPU and firing off GPU kernels on available GPU device on appropriate GPU supporting platforms. These actions form part of the host code which is executed on the CPU. An example of host code listing is shown below.

```
#include <CL/cl.h>
```

```
...
/*1. Get the Platform */
clGetPlatformIDs(1, platforms, &platforms_count);
/*2. Get the GPU devices, max 10*/
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 10, devices, &devices_n);
/*3. Create the CL context inside which compute devices will be available*/
clCreateContext(NULL, devices_count, devices, &pfn_notify, NULL, &_err)
/*4.  Create a command queue which will queue kernel calls */
command_queue = clCreateCommandQueue(context, device_id,CL_QUEUE_PROFILING_ENABLE,
    &ret);
/
/*5. Create a program from the kernel source */
cl_program program = CL_CHECK_ERR(clCreateProgramWithSource(contexts, 1,
       (const char **)&source_str, (const size_t *)&source_size, &ret));
/*6.  Build the program */
clBuildProgram(program, devices_count, devices, "", NULL, NULL);
/*7. Create the OpenCL kernels */
cl_kernel sum_kernel= clCreateKernel(program, "constant_add_gpu", &ret);

/*8.Create memory buffers on the device for the vector */
cl_mem lmv_mem_obj=clCreateBuffer(context, CL_MEM_READ_WRITE,
            (global_item_size)* sizeof(float), NULL, &ret);
/*9. Load data into the memory buffer*/
float lmv[global_item_size];
initialize(lmv);
clEnqueueWriteBuffer(command_queue lmv_mem_obj, CL_TRUE, 0,
             global_item_size* sizeof(float), lmv, 0, NULL, &buffer_completion);
/*10. Set kernel Arguments - num value */
int myNum=10;
clSetKernelArg(sum_kernel, 0, sizeof(int), &myNum));
/* 11. Now execute the kernel by enqueuing it on the NDRange ! */
clEnqueueNDRangeKernel(command_queue,sum_kernel, 1, &start_offset,
    &global_item_size,NULL,0,NULL,&kernel_completions);
/*12. Wait for computations to be over*/
clWaitForEvents(1, &kernel_completions);
/*13. Load the results back to CPU after GPU computation is over */
clEnqueueReadBuffer(command_queue, lmv_mem_obj, CL_TRUE, 0,
    global_item_size*sizeof(float), lmv], 0, NULL, &readEvent);
/*14. Finally clean the cl buffers*/
 clReleaseCommandQueue(command_queue); ....
```

As seen from the code listing, a lot of boiler plate code dealing with loading data to and from the GPU and executing the actual computation kernel needs to be written on the host side (steps 1 to 10 and 12 to 14).

In the case of simultaneous CPU-GPU processing, it is usual for the sequential control intensive tasks of an application to be executed on the CPU while the data parallel tasks of the application that need to be operated in a SIMT manner are executed by the GPU or accelerator.

2.2.4 Current Parallel Programming Model Advantages and Limitations

We now look at the comparative advantages and limitations inherent in these existing message passing, shared memory and accelerator models.

2.2.4.1 Message Passing

Advantages

Most high performance computing codes are written in the message passing paradigm. The advantages of employing message passing are :

- Distributed systems require explicit message passing models and the overall cost of building such clusters is less compared to large shared memory systems.
- Message passing models like MPI are more portable across systems as they can be utilized on both shared memory and distributed architectures
- Message passing models provide better safety because each process has its local copy of variables on which it operates. This prevents any kind of race condition or unexpected change of local variable state.

Limitations

MPI is a low level distributed programming model that has its share of drawbacks as mentioned in [94]. These are as follows:

- It places the entire burden on the programmer for being responsible for many low level implementation details, ranging from setting the communication among processes to resolving any deadlocks that may arise.
- The programmer alone has a global view of the problem as the processes themselves are only able to read and write data to their respective local memory. This data is copied and communicated to local memory locations of the processes through method calls making global operations expensive.
- Finally making the programmer responsible for all low level decisions increases the scope for adding bugs to programs.

2.2.4.2 Shared Address Space Programming

Advantages

- Shared address space models are easier to program and understand due to readily available loop parallelization constructs that require few lines of code to implement. This also allows maintainability and easier debugging.

- Shared address space models like OpenMP provide flexible parallel coding due to incremental parallelization obtained by adding directives over loops or blocks. The parallel parts of the code can run sequentially on a single core with no modification.

Limitations

In case of the Shared Address Space paradigm the ease of programming comes with some drawbacks. They are as follows:

- SAS systems face performance limitations due to poor spatial locality and high protocol overhead. This was observed in [92], where the GeNIMA SVM protocol was used for SAS on a cluster of eight, 4-way SMPs and it was found that using this in place of MPI gave roughly a factor-of-two deterioration in performance for many applications.

- While MPI has extra work upfront, optimization and debugging is easier compared to multi-threaded shared memory systems. So project time to solution can be longer in SAS [70].

- Unlike message passing systems, shared address systems need to deal with race conditions which can be challenging. For example, proving that a shared address space problem using semaphores will be race free, is an NP-complete problem [70].

2.2.4.3 Accelerator Programming

Advantages

- Accelerator programming languages like CUDA and OpenCL provide the only means to program GPUs for general purpose computing and offer optimized solutions for exploiting the concurrency from hundreds of GPU cores.

Limitations

- Accelerator programming languages suffer from the drawback of being very low level with the application programmer responsible for transforming data based on the memory layout of the accelerator device and handling data transfer between host and accelerator devices.

- The programmer has to learn domain specific libraries that target particular architectures. For example CUDA programming is aimed at NVIDIA based accelerators only. OpenCL is more generic than CUDA but it requires a lot more boiler plate setup code in comparison to CUDA as discussed in subsection 2.2.3.
- The SIMT programming model of CUDA and OpenCL provides flexibility in the way of expressive scalar syntax but has the drawback of maintaining registers to store redundant data items. The indirect memory access is also limited to registers so $a[b[i]]$ can scale to tens of elements, not tens of thousands of elements. This is because indirect memory access is unfeasible at the DRAM level (which is farthest from GPU cores, sitting outside the chip, see figure 1.1) since random access is not efficient and even within shared memory random access is slowed by bank contentions[61].
- Divergent flow leads to randomizing of memory access and unlike the SPMD (Single Program Multiple Data) model only one path is executed at a time forcing idle threads to wait for the active thread to complete execution. This makes multiple $if - else$ branches expensive and contrasts with pthreads whose SPMD model allows concurrent execution of multiple branches.

2.2.5 Summary

The drawbacks found in current parallel programming models suggest the need for a model that provides a layer of abstraction to handle low level operations while allowing the programmer to concentrate on the problem at hand. Also, the very existence of a multitude of parallel programming models makes integrating them to address heterogeneous parallelism a challenging task. This motivates the case for algorithmic skeletons.

2.3 Algorithmic Skeletons

A skeleton is essentially a second order function which accepts functions instead of simple data types as its arguments, and returns functions that accept the details of a specific problem as its results. The arguments passed to a skeleton are methods which are specific to a problem, while the structure of the skeleton would be the overall computation pattern that needs to be applied to the methods passed to it.

Advantages

- Algorithmic skeletons address the two fundamental issues of any parallel programming model, i.e. of *problem decomposition*, which is the identification of parallelism, and *distribution*, which is the physical implementation of the parallelism identified by decomposition.

- The distribution and manipulation of data across the processors based on the hardware topology is also handled implicitly by the skeleton. The application programmer has to be only concerned with supplying the arguments and may not even be aware of the implicit parallelism built into the skeleton. The system implementer creates the general computational structure of the skeleton.
- The fragmented programming nature of skeletons allows independent skeletons to co-exist, differentiating them from other systems that are either too highly abstract to implement any specific solution, or are restricted to function optimally in a particular hardware setting.

Limitations

- There is a restriction placed at the highest level of the structure of the skeleton regarding what computation it can perform. Since the programmers are not aware of the underlying parallel implementation (and they do not have access to it), they are restricted to exploiting parallelism offered by the skeletons without being able to modify the skeletons.
- The restricted structure of skeletons means they are generally not portable across different architectures without tuning some of their components. This is elaborated later in section 2.3.3.

2.3.1 Classifying Skeletons

Poldner et al. [54] have classified Skeletons as either Task Parallel or Data Parallel depending on the kind of parallelism used. Task Parallel skeletons are responsible for dynamically setting up the communication among processes and distributing tasks using the concept of nesting. They include the farm or pipeline skeletons [20] among others. Task parallel skeletons, like atomic processes, accept a sequence of inputs and produce a sequence of outputs. This allows them to be nested in any manner. A concrete example of the task parallel skeleton, the Task Farm is shown in figure 2.6. The farmer node distributes tasks to worker nodes in a demand driven manner and then collects results from the workers. Task Farms can have multiple farmer nodes handling many workers, and computation involves an iterative process of the farmer distributing a large task across idle worker nodes, receiving results and assigning the next piece of task and so on, until all tasks are complete. The Data Parallel Skeletons, in contrast to the task parallel ones, perform the same task or operation across a bulk data structure such as an array. Examples of this type of skeleton are Map, Fold, Rotate [25] and the wavefront which will be discussed later in section 2.5.1.

Abstractions are also provided by Intel's Threading Building Blocks (TBB) [31]. This has become popular for multi-core processors by providing easy to use thread safe container

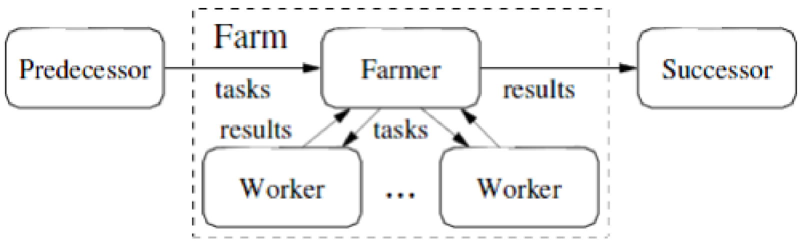


Figure 2.6: The Basic Task Farm, taken from [54]

classes and templates, that abstract out low level details such as multiple thread synchronization and load balancing among processes. In Threading Building blocks skeletons are found in the algorithm templates like pipeline and parallel _do.

With this overview of different types of algorithmic skeletons we now focus on some practical guiding principles for the design and development of skeletal systems.

2.3.2 Guiding Principles for Skeleton System Design

Research into design patterns for high performance computing stresses the advantages of Object Oriented (OO) languages for implementing skeletons [21]. The convergent experiences of design pattern based and skeleton based systems necessitate sound software engineering principles and motivate four principles which we will follow in our design.

- **Minimal conceptual disruption** - Skeletal programming is neither a completely abstract declarative system based on functional programming nor is it a fully object-oriented one. It should not be restricted to these two concepts rather it should be a bridge to the de facto standard for parallel programming.
- **Integrate ad-hoc parallelism** - Since we cannot expect skeletons to provide complete parallelism, the system should have a well defined process to allow integration of skeletons with ad-hoc parallelism.
- **Accommodate diversity** - The system should not be restricted by its own specification. For example, a pipeline specification requiring each stage to produce one output for each input is too restrictive, as it excludes algorithms that have stages where one input generates many outputs or none at all.
- **Show the payback** - As per the cost benefit analysis, the benefits of adopting a new technology should outweigh the costs. The skeletal programs must outperform conventional implementations with equivalent investment of effort in developing them and they should be easily ported to new architectures with minimal changes to source and with sustained performance.

We have followed these guidelines in designing our wavefront framework. We provide minimal conceptual disruption by providing a C++ library with method and parameter declarations in wavefront terminology (explained in subsection 2.5.2). These can be overridden with custom defined data structures and kernel functions. The methods can also be plugged into existing code, allowing ad-hoc parallelism. We accommodate diversity by providing support for 2D and 3D wavefronts, along with choice of dependent elements up to two preceding diagonals. The payback is demonstrated from the performance gains of employing our autotuning framework.

With this brief discussion of the guiding principles behind designing and implementing skeletons, we now look at the limitation of skeletons regarding performance portability.

2.3.3 Tuning of Skeletons

Skeletons solve the challenge of problem decomposition and distribution but they don't readily address the issue of performance portability when moving from one architecture to another. This can be illustrated by the task parallel Task Farm. In a task farm, there are several opportunities for optimizations that include

- Choosing the correct number of tasks to transfer in a single farmer-worker interaction, i.e. chunks of tasks being communicated.
- Selecting the correct number of workers to prevent bottleneck at either the farmer or worker's end, depending on the size of task.
- Adjusting the size of the data that is being communicated for each task to minimize communication overhead.
- Choosing between single and multiple distributed farmers.

All these parameters may have some impact upon the overall performance by reducing the effect of bottlenecks. They have different optimal settings for different combinations of architecture and the farmed application. These combinations of various input features can, and do produce an enormous number of possibilities.

One approach to exploring this space efficiently is to apply machine learning [74] which is introduced in next section.

2.4 Machine Learning

Machine Learning refers to the field of study in computer science and statistics of algorithms that improve from experience without any explicit programming instructions. A formal definition of machine learning in operational terms is provided by Tom Mitchell [74] as "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." Based on this definition we can say the purpose of a machine learning algorithm is to improve performance on new unseen tasks after experiencing tasks from a training data set. This training data set consist of values of input features or attributes that define a problem and is representative of the space of occurrences of those problem instances. The learning model trained on this data has to be then general enough to predict desired output values for unseen problem instances with high accuracy.

Machine learning algorithms are classified on the basis of the type of input feature set available during training and the type of desired outcome. When training instances are labeled and the objective is to infer a function by analyzing the instances, it is called *supervised learning*. For evaluating the accuracy of a supervised learning model, the predicted values are either validated against known best values of the unseen instances or cross validated against a subsection of the training data, called the test validation set, which contains instances that were not part of training. We have used supervised learning techniques in our thesis and examples of those will be discussed in detail in 2.4.3. On the other hand, in *unsupervised learning* there are no labeled instances of training data since the objective is to find patterns in data or the underlying hidden structure. This also means there is no known solution against which error or reward signals can be used for validation. Examples of unsupervised learning include k-means clustering [49] and hidden Markov chains [103]. If clustering algorithms create meaningful classification, then evaluation can be done through creation of external data sets by manual hand-labeling and validating the accuracy. Thus they can be evaluated using internal metrics information on the computed clusters when the clusters are clearly demarcated or by using external metrics that carry statistical testing on the structure of the data [49].

2.4.1 Need for Machine Learning

The two conventional alternatives to machine learning are hand tuning and analytic modeling. One advantage of hand tuning is the greater level of optimization typically possible since the code is written to exploit specific architecture and application. However it is very time consuming and hand tuned code is unlikely to be performance portable. Analytic models (some of which are explored in section 7.2 and section 7.4) can provide an alternative to machine learning based tuning. These use mathematical equations consisting of application and architecture specific variables that are easily ported across systems. However, coming up with a good analytic solution that can cover a wide range of system and application variations while providing high accuracy is quite challenging. Besides, when the source code is not accessible to the end user, such as when using a third party library, the analytic model cannot be constructed. For such black box systems, autotuning by learning from experience is often the only solution.

2.4.2 Machine Learning in Program Optimization

We present an example which illustrates how machine learning can be useful in software optimization. Let us consider image processing. It is very hard to manually write a program to recognize a particular image or a face. Instead, we can collect many samples that correspond to the correct image for a given set of input features such as texture, color, sharpness etc. A machine learning algorithm can take these example image instances and produce a program that predicts the correct image for a set of input features, to some degree of accuracy.

Similarly, machine learning techniques can be applied to a simple task farm skeleton with a single farmer and multiple workers to predict the correct number of workers for numerous combinations of farm input features that result in optimal efficiency. Like the image processing example, it is not possible to tune the parameters of our farm without having prior knowledge or obtaining the prior inductive bias [99] from lots of training instances that are subsequently fed to machine learning methods. Inductive bias refers to the assumptions made in a machine learning model to predict outputs for previously unseen input data.

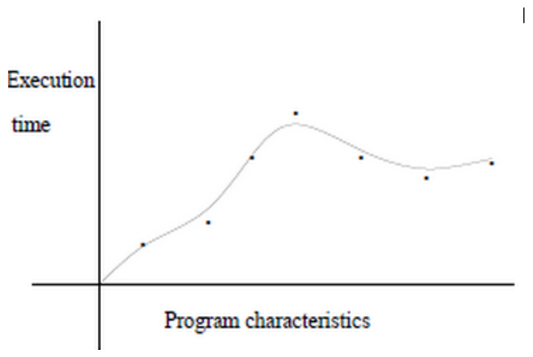


Figure 2.7: Learning as Curve Fitting

In general, we can conceptually define machine learning as a sophisticated form of curve fitting where the inputs are characteristics of the program, and architecture and the outputs are the optimization function we are interested in, such as runtime of the program. After fitting the learning model to the data consisting of known values of input and output parameters, the model can predict the value of desired output parameters such as the number of workers in a task farm skeleton. These values must then be cross validated to check the accuracy of the prediction. Once the model is defined, we predict future behavior and find the best optimization for a set of input features.

Having made the case for using machine learning, we now provide an overview of the specific learning models that we have used.

2.4.3 Machine Learning Models in Supervised Learning

Machine learning applications can range from predicting discrete labels, i.e. - classification tasks [37] such as classifying stars or recognizing handwritten/spoken words, recognizing ham from spam in emails etc, to predicting real numbers such as predicting future stock prices or exchange rates and monitoring credit card transactions among others. We are specifically interested in supervised learning [37] which deals with predicting an output y depending on an input x as in $y = f(x)$. Supervised learning creates this predictor $f(x)$ from training data D on the basis of some prior bias. The importance of bias can be summarized from Mitchell's statement [74] as *"a learner that makes no a priori assumptions regarding the target concept has no rational basis for classifying any unseen example."* This bias is inductive as generalizations are made about the form of $f(x)$ based on instances of different supervised learning methods corresponding to different inductive biases. The ones used in our thesis include the Linear Regression model, SVM and M5 pruned decision tree (M5P) [37].

The general structure of such supervised learning methods given by Hand et al [26] is

- Define the task - in our case it is predicting a whole number (we round decimals to nearest whole numbers)
- Decide on the model structure - the choice of inductive bias (mentioned in subsection 2.4.2), we have used Linear regression, SVM and M5P.
- Decide on the score function to judge the quality of our curve fitting - for example, squared error used in Linear Regression and so on
- Decide the optimization/search method to optimize the score function.

Now we discuss the three models used in our work starting with Linear Regression.

2.4.4 Linear Regression

Linear Regression models the relationship between a scalar output variable and one or more input variables using linear functions. The simplistic curve fitting model (figure 2.7) stated earlier in this section is an example of this. Linear regression is used to fit a predictive model to an observed data set of y and X values where $X = x_1, x_2, ..., x_n$, i.e. set of input features. After developing this model, for an additional value of X, the model predicts the value of y. The score function here is the squared error or likelihood given by the equation

$$E(w) = \sum_{i=1}^{n} (y_i - f(x_j; w))^2 \tag{2.2}$$

where w represents the weight associated with each input attribute.

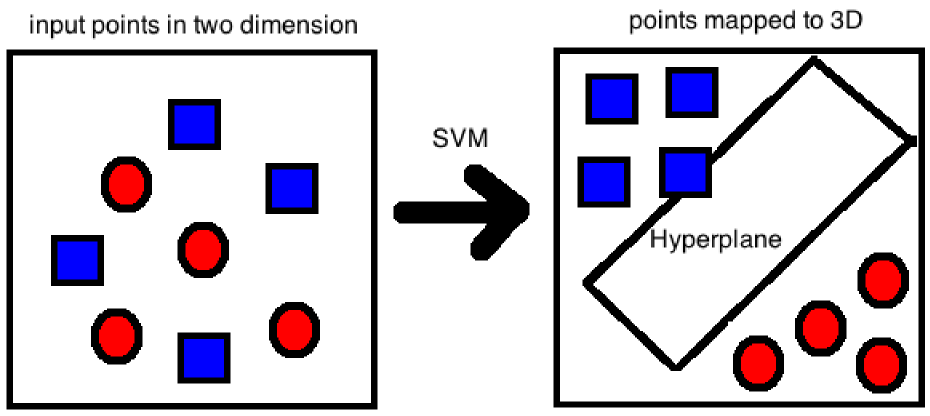


Figure 2.8: Support Vector Machines - Mapping the points in 2D to 3D in-order to be separated by a hyperplane for classification

2.4.5 Support Vector Machines

The next model is the Support Vector Machine (SVM) which maps the input data into a higher-dimensional space and then finds a hyper-plane in this space which separates the data. This is because while the original problem may be stated in a finite dimensional space, it may not be possible to linearly separate the points in that space into different sets. However, if the original space is mapped onto a much higher-dimensional space as illustrated in figure 2.8 then the points can be separated for classification, regression or any other task.

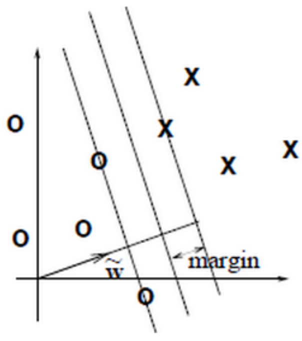


Figure 2.9: The Support Vector Machine model, adopted from [32].

This mapping can be done either using a kernel function or a radial basis function. There are two parameters, gamma and cost, that control the SVM's behavior and tune its accuracy.

If we consider a two class example (+1 and -1) with linearly separable training data, we can select two hyper-planes that separate the data such that there are no points between them. We then try to maximize their distance. The region bounded by them is the "margin". Mathematically, if the perpendicular distance from a hyperplane in the bounded region to the nearest +1 class point is d_+ and similarly d_- for the nearest -1 class point, the *margin* is defined as $min(d_+, d_-)$. We want to find the maximum-margin hyperplane that divides the

+1 class points from the -1 class points. The support vector machine algorithm looks for this maximum *margin*, i.e. choosing a hyperplane so that $d_+ = d_-$ as shown in Figure 2.9.

The hyperplanes themselves are described by the equations

$$\mathbf{w} \cdot \mathbf{x} - b = 1 \tag{2.3}$$

and

$$\mathbf{w} \cdot \mathbf{x} - b = -1 \tag{2.4}$$

This way of separating data makes SVM a discriminative learning model which focuses on class boundaries. The linear SVM usually performs better than the linear regression model owing to its default policy of normalizing training data and, apart from polynomial kernels it can also use radial basis functions to handle non linearity,

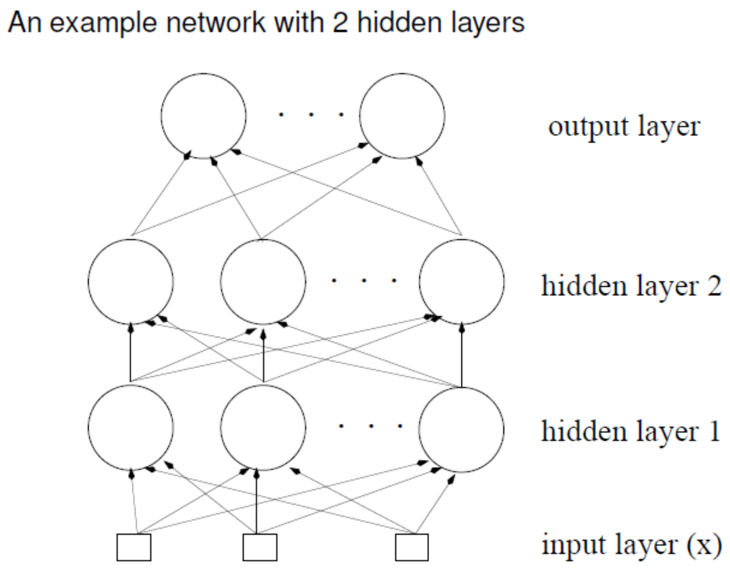


Figure 2.10: The Multi layer Perceptron Model, adopted from [32]

2.4.6 Multi Layer Perceptron

Multi Layer Perceptrons (MLPs), illustrated in Figure 2.10, are a form of artificial neural networks [37]. They consist of multiple layers of nonlinear activation functions that transform the data input to obtain the output. Linear activation functions can be used to make a regression model, whereas a logistic activation function such as

$$g(z) = 1/(1 + e^{-z}) \tag{2.5}$$

or the Gaussian function can be used to solve classification programs. There can be an arbitrary number of hidden layers and each unit in the first hidden layer computes a non-linear function of the input x. Each unit in a higher hidden layer computes a non-linear function of the outputs of the layer below.

MLP is a powerful function approximator but it incorporates overfitting due to repeated back propagation of errors resulting in small training error but large validation error. Techniques to improve generalization in multilayer perceptrons are discussed in [88]

2.4.7 Decision Trees

Decision Trees can be used for either supervised classification or regression. A decision tree is composed of the root node that branches out into subsequent parent nodes that terminate at leaf nodes. In the case of classification, the leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Thus each non leaf node represents a condition based on which decisions are taken leading to further nested conditions or termination at leaf nodes which represent outcomes. This is illustrated in figure 2.11 where the conditions are Outlook=Overcast, Humidity=Normal and Wind=Weak.

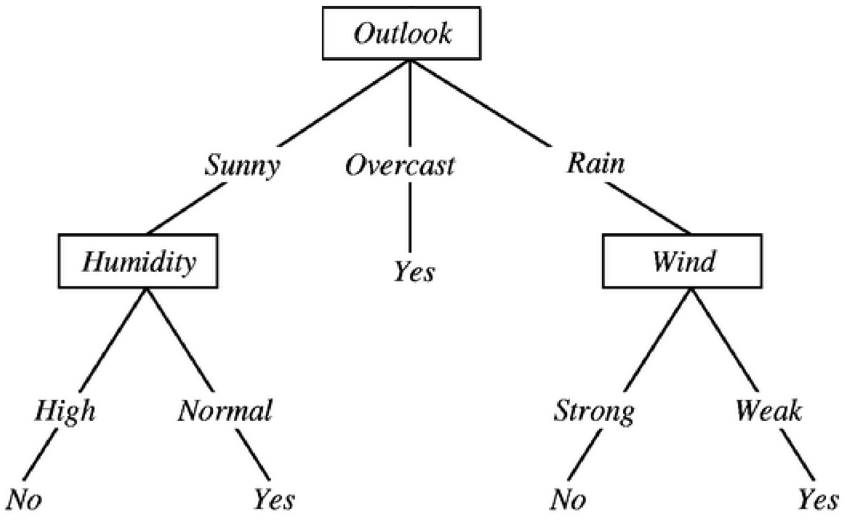


Figure 2.11: The classification Decision Tree, adopted from [74]

For regression, each leaf node calculates a real number which is the predicted outcome of the output attribute. All non leaf nodes are conditions against which real valued input attributes are tested and based on the result, each non leaf node branches into subsequent nodes or it terminates at a leaf node. We use Ross Quinlan's M5 algorithm [87] which is a regression tree algorithm where linear regression equations on the leaves calculate values of the output attribute we are interested in. Moreover, the M5P algorithm allows the generation of pruned or unpruned trees. Pruning is required because, as with MLP, decision-tree learners can create over-complex trees that do not generalize well from the training data leading to over-fitting. Increasing the number of training instances and applying techniques like k-fold cross validation improves accuracy of prediction as k-fold cross validation combats overfitting [78]. However, there is a limit to improving the accuracy of tree based learners through increased numbers of training examples. This can be seen from Figure 2.12 where increasing the number of examples increases the tree based learners accuracy up to a limit while a naive bayes learner actually overtakes the tree learner.

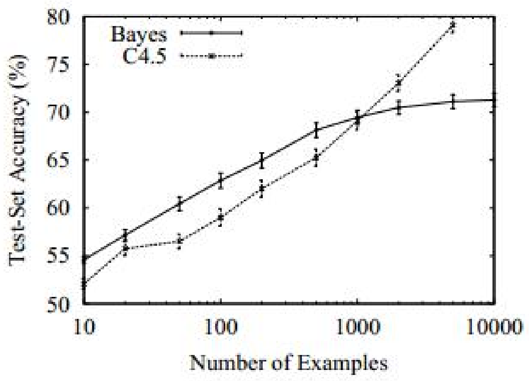


Figure 2.12: This figure illustrates how the naive bayes outperforms the tree based learner whose accuracy saturates after a threshold amount of examples is reached, adopted from [29]

In the context of our work, once a learning model has been trained to a sufficient level of accuracy, it can be utilized after compile time profiling of a program to automatically tune the skeleton feature space. Tuning would lead to an improvement in run time performance and reduction in parallel cost.

Having made the case for applying machine learning we now provide the background details of the data parallel skeleton that forms the core of our thesis, the wavefront. We begin with a general introduction to Dynamic Programming Problems [13] as the wavefront pattern belongs to this class.

2.5 Dynamic Programming Problems

Dynamic Programming is the process of dividing complex problems into manageable overlapping sub-problems, whose solutions are reused to solve other sub-problems. The wavefront pattern is a class of Dynamic Programming problems. A Dynamic Programming problem is expressed as a *functional equation* which is a recursive equation that represents the solution, with the left side being the unknown quantity to be computed and the right side being an aggregate function such as a min, max, average etc [13]. Each Dynamic Programming problem can be classified by its properties as :

- Monadic : The functional equation contains one recursive term per subproblem.
- Polyadic : The functional equation contains multiple recursive terms per subproblem.
- Serial : The sub problems depend on previously computed results from only the immediately preceding stage.

- Non-Serial : The sub problems depend on previously computed results from many previous stages.

Thus dynamic programming problems belong to one of the four categories which we briefly discuss with examples for each case.

- **Serial Monadic** : The 0/1 Knapsack problem belongs to this class. Given a knapsack of capacity C and weight of each of n objects being w_i, the goal is to maximize the sum of the knapsack profit p_i subject to the constraint of capacity of knapsack. Objects inside the knapsack are denoted by $v_i = 1$ and those outside the knapsack have $v_i = 0$. This can be expressed mathematically as

$$\sum_{i=1}^{n} w_i * v_i \leq C \tag{2.6}$$

$$\max(\sum_{i=1}^{n} p_i * v_i) \tag{2.7}$$

These conditions are realized in the functional equation where the unknown quantity in the left hand side $F[i,x]$ is the maximum profit for a knapsack of capacity x using objects $1,2,...,i$. The solution is realized as

$$F[i,x] = \begin{cases} 0, & x \geq 0, i = 0 \\ -\infty, & x \leq 0, i = 0 \\ \max(F[i-1,x],(F[i-1,x-w_i]+p_i)) & 1 \leq x \leq n \end{cases}$$

In the first case maximum profit for a knapsack of some capacity is 0 if there are no objects available. The second case excludes a knapsack of negative capacity. The third case recursively maximizes the profit at position $[i,x]$ by selecting the maximum profit from the preceding level where inclusion of an object i with weight w_i reduces capacity by that amount and increases profit by p_i.

There is only one recursive term in the equation, making it monadic, and as seen from figure 2.13, each node depends on two subproblems at preceding level only, thus making it serial.

- **Non Serial Monadic** : The *functional equation* of the Longest Common Subsequence or Smith Waterman algorithm [95] belongs to the Non Serial Monadic variety. It is a string alignment problem in bio-informatics that finds the global maximum of the longest matching sub-strings between any two genetic sequences. Given any two DNA sequences $A =< T,A,T,G,A,C,... >$ and $B =< G,C,A,T,G,A,... >$ of size m and n respectively, the goal is find the longest sequence that is subsequence of both A and B (here it is $< A,T,G,A >$). The *functional equation* is given below where $F[i,j]$ is the

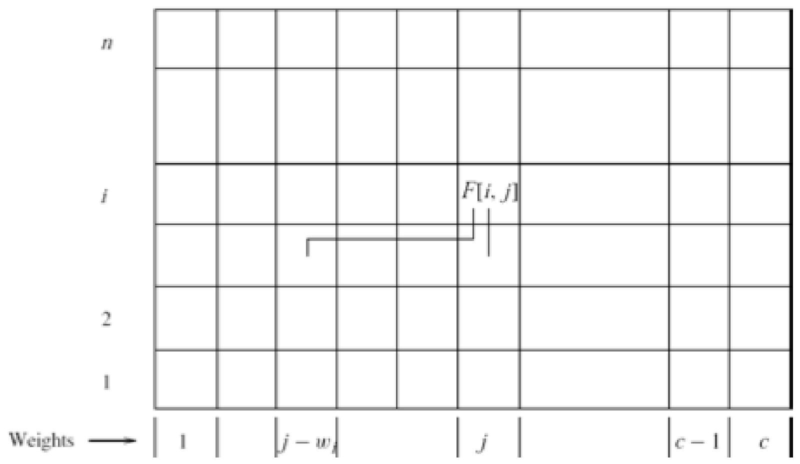


Figure 2.13: Serial Monadic Dynamic Programming problem with one recursive term and dependent on subproblems at preceding level only, adopted from [48]

length of the longest common subsequence for the first i elements of A and the first j elements of B. The goal is to determine $F[m,n]$, i.e. find the longest subsequence over the entire lengths of A and B.

$$F[i,j] = \begin{cases} 0, & i=0, j=0 \\ F[i-1,j-1]+1 & i,j \geq 0; x_i = y_i \\ \max(F[i,j-1], F[i-1,j]) & i,j \geq 0; x_i \neq y_i \end{cases}$$

Here there are two recursive terms but it is monadic as they are in different subproblems. Each node in a diagonal depends on two subproblems in the preceding level (west and north of the element) and one subproblem two levels earlier (north-west position), making it non serial. This is shown in figure 2.14.

- **Serial Polyadic** : Floyds all pair shortest path algorithm[56] is an example of serial polyadic dynamic programming problem. It has the two recursive terms in the second sub-problem making it polyadic and each node depends only on the preceding level sub-problems being computed making it serial. Its functional equation is shown below

$$d_{i,j}^{k} = \begin{cases} c_{i,j} & k=0 \\ \min(d_{i,j}^{k}, (d_{i,k}^{k-1} + d_{k,j}^{k-1})) & 0 \leq k \leq n-1 \end{cases}$$

  The dependency relation can be seen in figure 2.15

- **Non Serial Polyadic** : The Optimal Matrix Parenthesization problem [48] belongs to this class of problems since its *functional equation* has two recursive terms in each sub-problem making it polyadic and each node depends on previously computed sub-problems from more than the preceding stage, making the problem non serial. The dependency relation can be seen in figure 2.16

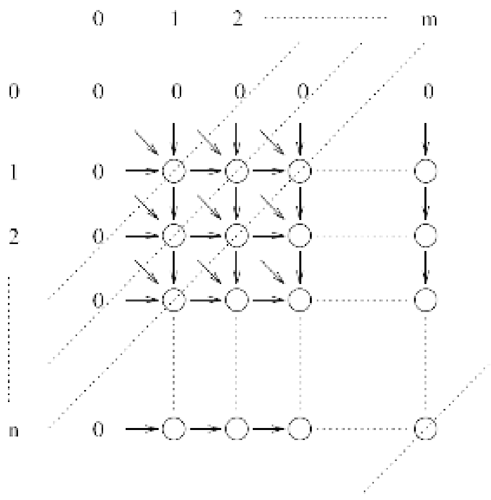
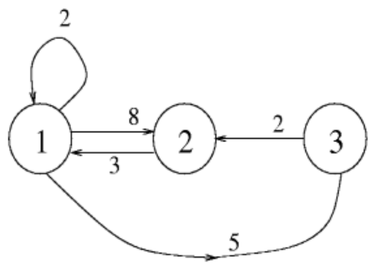


Figure 2.14: Non Serial Monadic Dynamic Programming problem with one recursive term but dependent on two subproblems at preceding level and one problem two levels up, adopted from [48])



$$C = \begin{bmatrix} 2 & 8 & 5 \\ 3 & \infty & \infty \\ \infty & 2 & \infty \end{bmatrix}$$

$$D_0 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}; \; D_1 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{bmatrix}; \; D_2 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}; \; D_3 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

Figure 2.15: Serial Polyadic Dynamic Programming problem with two recursive terms (C and D) and dependent on two subproblems at preceding level, adopted from [56])

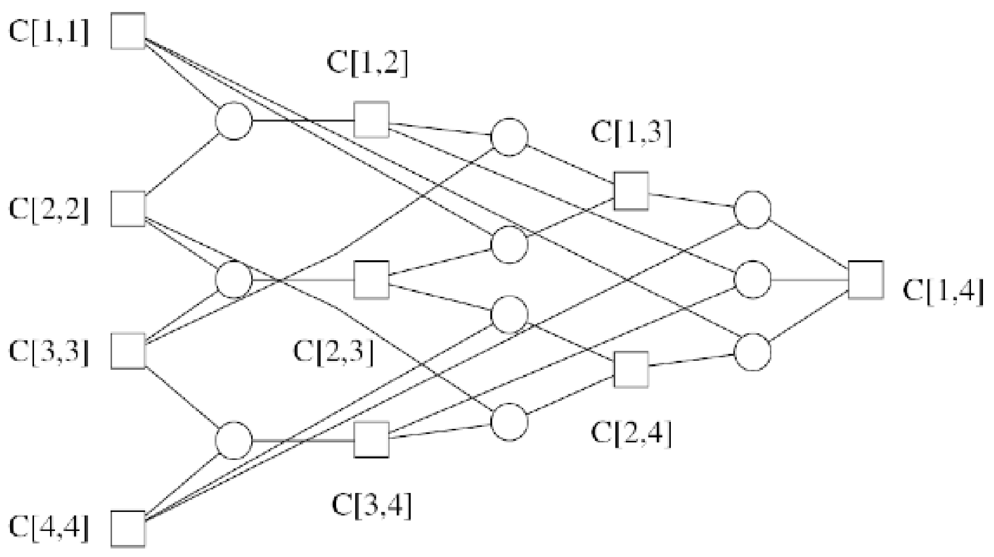


Figure 2.16: Non Serial Polyadic Dynamic Programming problem with two recursive terms and dependent on subproblems which are located further than the preceding level, adopted from [48])

$$C[i,j] = \begin{cases} 0, & j \geq 0, 0 \leq i \leq n \\ \min_{i \leq k \leq j}(C[i,k] + C[k+1,j] + r_{i-1}r_k r_j) & 1 \leq i \leq j \leq n-1 \end{cases}$$

The Non Serial Monadic class of Dynamic Programming problems form the basis of our research, with the wavefront patterns being the prime examples. We earlier examined one example of the wavefront - the longest common subsequence application. Other examples of wavefront applications are discussed in detail in section 2.6. In the next subsection we introduce the wavefront and discuss the general terminology associated with it.

2.5.1 Wavefront

The wavefront pattern [4] abstracts computations which evaluate a class of multi-dimensional recurrence relations. The values of the relation are computed into a multidimensional array. Figure 2.17 gives a graphical representation of a two-dimensional wavefront in which computation starts from a corner of the grid at position (0,0). This computation propagates to neighboring elements in a series of diagonal bands, resulting from the dependencies inherent to this pattern. This wave-like sweep of computation gives the pattern its name. All elements in a diagonal can be computed simultaneously allowing scope for parallel computation as they only depend on their previous diagonals being computed. In figure 2.17, the number of elements that can be computed in parallel grows from 1 in iteration 0 to a maximum of 4 for elements in position $(3,0),(2,1),(1,2),(0,3)$ in iteration 3, positions $(3,1),(2,2),(1,3),(0,4)$ in iteration 4 and positions $(3,2),(2,3),(1,4),(0,5)$ in iteration 5. It then decreases to 1 by iteration 8. Thus in a wavefront pattern, the maximum number of elements that can be

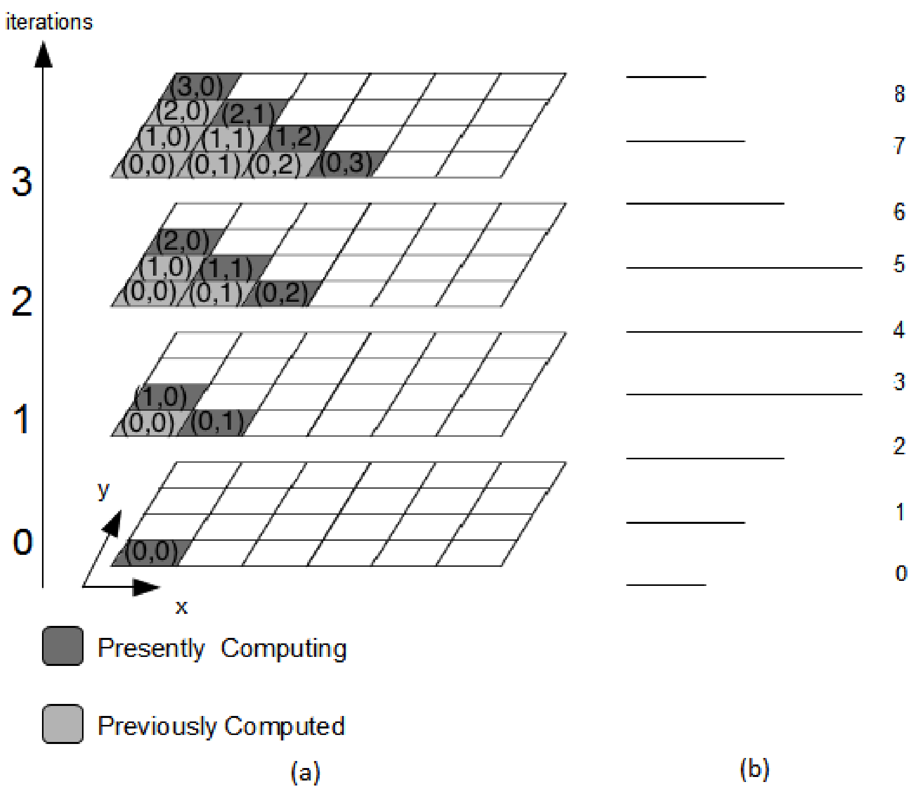


Figure 2.17: (a) Waveflow for a two dimensional instance of size 4 x 6 (b) The number of concurrently computable elements increases from iteration 0 until maximum parallelism is achieved at iterations 3,4 and 5. Part (b) of the figure is inspired by [1].

computed in parallel are found in the middle of the computation grid, narrowing down as we move away from the longest diagonals. We now present terminology associated with the wavefront pattern.

2.5.2 Wavefront Terminology

For our purposes, the key characteristics of a wavefront instance are as described in table 2.1.

The number of rows in the array are represented by *dim*. For simplicity we assume square arrays, but this restriction could be lifted straightforwardly. The granularity of the computation at each point in the array, which we assume to be regular as is typically the case, is captured by *tsize*. The number of floating point data items at each point in the array, providing a measure of data granularity, is represented by *dsize*. The neighboring elements of each point being computed are referred by their relative positions such as *north*, *west*, *northwest* and *top*. These characteristics form the input parameters to our autotuning framework. Their experimental values are discussed later in chapter 4, chapter 5 and chapter 6. In the next section we discuss the various wavefront applications that form part of

Table 2.1: Input Parameters

Parameter	**Description**
dim	Overall problem size expressed as width of the array. For simplicity, we deal with 2D square matrices or 3D cubes.
tsize	Measure of granularity of the task that accounts for the time taken per element computation. In our experiments, low values (<10) indicate per element computation is in milliseconds while high values (>5000) indicate per element computation takes few seconds
dsize	Measure of element data granularity that is based on the number of floating point data types and has implication for time spent in transferring data between accelerators and CPU cores.
north	In a 2D wavefront, north is the element at (i-1,j) position for an element at position (i,j)
west	In a 2D wavefront west is the element at (i,j-1) position , for an element at position (i,j)
northwest	In a 2D wavefront northwest is the element at (i-1,j-1) position , for an element at position (i,j)
top	In a 3D wavefront, top is the element at (i,j,k-1) position for an element at position (i,j,k)

our experimental setup.

2.6 Wavefront Applications

For our research we selected two applications that are 2D wavefronts and one 3D application. Our criteria for choice were based on contrasting size, task and data granularity. We also chose these applications because they have uses across diverse domains, ranging from finance to nuclear physics. One 2D application, the Longest Common Subsequence, is computationally fine grained with per element computation taking few microseconds and the amount of data transferred limited to a few bytes. It also has the largest problem size among all three wavefront applications. This application is relevant in bio-informatics for gene sequence matching. The second 2D wavefront application is the Nash Equilibrium. It is a coarse grained with per element computation in the range of hundreds of milliseconds to few seconds and the data transferred being tens of bytes, but its problem instances are smaller in size. This application is relevant to financial game theories. The 3D wavefront application is a modified subroutine from the Lower-Upper Triangulation application which is an important linear algebra problem. It has granularity that lies in between the 2D applications but has a higher number of data elements. The contrasting attributes of these applications provide interesting test cases for our auto-tuning framework and are representative of a range of wavefront application classes. We now discuss these applications in detail.

2.6.1 Nash Equilibrium

In game theory, a group of players in zero-sum game are in equilibrium if each player takes the best decision taking into account decision of every other player. This can be applied to a game of market share of a product in a zero sum game, where increase in market share of one company results in decrease of its competitors. Such a case involves a company taking the best decision by taking into account the marketing strategies of other companies manufacturing that product. Thus in a a non-cooperative game involving two or more players, if each player is assumed to know the equilibrium strategies of the other players, and no player has anything to gain by changing only their own strategy, then they are in Nash Equilibrium [104].

Nash Equilibria also exists in coordination games. A simple example is the stag hunt game in which two players may choose to hunt a stag that provides more meat (4 utility units) or a rabbit (1 utility unit) with the limitation that if one player hunts a stag while other hunts a rabbit, the stag hunter gets 0 utility units. However, if both hunt stag, the payoff is split evenly between both (2 utility units each). Thus Nash Equilibrium exists at (stag,stag).

Table 2.2: Payoff Matrix for a Cooperation game in Nash Equilibrium

	Player 1 hunts Stag	Player 1 hunts rabbit
Player 2 hunts Stag	(2,2)	(1,0)
Player 2 hunts Rabbit	(0,1)	(1,1)

Even (rabbit,rabbit) which is less than the optimal payoff, is at Nash Equilibrium. Here neither player has any incentive to change strategy due to a reduction of payoff from 1 to 0. The payoff matrix is shown in table 2.2.

A formal definition of Nash Equilibrium is provided in [66]. In a game let $i = 1, 2, 3...I$ be the set of players. Let $S = S_1 \times S_2 \times ...S_n$, be the set of strategies for all players that fully specify all possible actions and u_i be the payoff function for each player. Let σ_i be the strategy profile of player i. Similarly let σ_{-i} be the strategy profile of all players except i. Then, a strategy profile $(\sigma_1, ..., \sigma_I)$ is a Nash equilibrium if no unilateral deviation in strategy by any single player is profitable for that player, that is

$$\forall i, s_i \in S_i : u_i(\sigma_i, \sigma_{-i}) \geq u_i(s_i, \sigma_{-i}) \tag{2.8}$$

The Nash Equilibrium application is characterized by small instances but computationally demanding kernel.

2.6.2 Longest Common Sub-sequence

This is essentially a string alignment problem from Bioinformatics [95] called the Smith-Waterman algorithm. It performs local sequence alignment for determining similar regions between two strings representing nucleotide or protein sequences. Instead of looking at the whole sequence, the Smith-Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure. This problem is characterized by very large instances and very fine-grained kernels, varying with detailed comparisons made. This application was earlier discussed as an example of the Non Serial Monadic category in section 2.5

2.6.3 Lower Upper Triangulation

In linear algebra, triangulation is a process of finding a lower or upper triangular matrix whose product is a given matrix [101].

A lower triangular matrix or left triangular matrix has the form

$$L^* = \begin{bmatrix} l_{1,1} & 0 & \dots & \dots & 0 \\ l_{2,1} & l_{2,2} & & & \vdots \\ l_{3,1} & l_{3,2} & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \dots & l_{n,n-1} & l_{n,n} \end{bmatrix} \tag{2.9}$$

and analogously an upper triangular matrix or right triangular matrix has the form

$$U^* = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \dots & u_{1,n} \\ \vdots & u_{2,2} & u_{2,3} & \dots & u_{2,n} \\ \vdots & & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & u_{n-1,n} \\ 0 & \dots & \dots & \dots & u_{n,n} \end{bmatrix} \tag{2.10}$$

The application consists of a technique called successive over relaxation (SSOR) which is a variant of the Gauss-Seidel method for solving a linear system of equations with fast convergence. SSOR can be understood with an example of a square system of n linear equations with unknown x. If these linear equations are represented as

$$A\mathbf{x} = \mathbf{b} \tag{2.11}$$

A can be decomposed into a diagonal component D, strictly lower component L and strictly upper triangular U.

$$A = D + L + U \tag{2.12}$$

The strictly lower-upper triangular matrices differ from those in equations 2.9 and 2.10 respectively with elements at their diagonals set to zero, i.e. if $L^*_{i=j} = 0$ and $U^*_{i=j} = 0$ then $L = L^*$ and $U = U^*$. The diagonal component is given by

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \tag{2.13}$$

The system of linear equations may be rewritten as:

$$(D + \omega L)\mathbf{x} = \omega\mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x} \tag{2.14}$$

for a constant $\omega > 1$, called the relaxation factor. By taking advantage of the triangular form of $(D + \omega L)$, the elements of $x^{(k+1)}$ are computed using forward substitution:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right), \quad i = 1,2,\dots,n. \tag{2.15}$$

which is a recurrence relation of the Non Serial Monadic variety of dynamic programming problem. The SSOR kernel used in our experiments comprises multiple sweeps until the solution converges to the desired value. The subroutine itself is composed of a Jacobian stencil operation followed by the lower-upper triangulation, which is the wavefront operation. There are also a significant amount of residual computations outside the SSOR computation.

2.7 Parallel Computing Metrics

For homogeneous parallel computing systems it is conventional to measure judge performance in terms of cost, speed-up and efficiency.

The cost of a parallel algorithm is the product of its run time T_p and the number of processors p. This cost translates into the real world financial cost of purchasing and maintaining hardware (nodes). Thus a small speed-up, achieved at higher cost, might not be attractive to the user.

For cost optimality, its cost must match the run time of the best known sequential algorithm for the same problem given by T_s. The relative speed up offered by a parallel algorithm is then the ratio of the run time of the sequential algorithm to that of the parallel algorithm. The equation for relative speedup is given by :

$$S = T_s/T_p \tag{2.16}$$

Linear speedup or ideal speedup is obtained for $S = p$. In an algorithm with linear speedup, doubling the number of processors doubles the speed. But ideal speedup in a parallel program is limited by Amdahl's law [53] , i.e. by the time needed for the sequential fraction of the program.

Another metric of performance is the efficiency E, given by the ratio of the speedup to the number of processors used.

$$E = S/p = T_s/(p * T_p) \tag{2.17}$$

However, these definitions of cost, speed-up and efficiency are specific to homogeneous systems and measuring performance by these metrics is problematic for heterogeneous systems of the type we employ. Our most complex systems involve multiple CPU cores, and also multiple GPU accelerators, each with multiple cores of fundamentally different capabilities and clock speed from the CPU. Rather than generating more complex speed-up and efficiency definitions, we proceed as follows: for each application and system, we find the best absolute performance point (i.e. tuning) obtained by a multicore CPU implementation alone, which may involve one or more cores, and compare the execution time of other heterogeneous implementations to this, and to the best performance point found in the whole heterogeneous search space. This allows us to see how much we are gaining by introducing heterogeneity, and to understand how close to optimal (with respect to our entire search space) our heuristics are.

2.8 Conclusion

In this chapter we discussed the necessary background information related to the need for algorithmic skeletons to address the parallel programming crisis. We then provided background details regarding tuning of skeletons through machine learning to enable performance portability

We described the wavefront pattern which forms the basis of our work, and introduced a number of wavefront structured applications. We concluded with a brief overview of parallel computing metrics.

Chapter 3

Wavefront Implementation Strategy

3.1 Introduction

In keeping with the principles of pattern-oriented parallelism, our wavefront framework simply requires the programmer to specify the types required to describe the data, initialization of any boundary values, and the core computational kernel which calculates a data-point as a function of its neighbors. The kernel is written in the OpenCL compliant subset of C. All other code, including parallelization, creation, initialization and distribution of data structures and all OpenCL machinery is hidden and tuned by our framework.

Our parallel wavefront execution strategy includes support for multi-core CPU parallelism, CPU tiling, GPU tiling, and the use of multiple GPUs. We first investigate the two dimensional wavefront implementation for single GPU and dual GPU architectures in subsection 3.2.1 and subsection 3.2.2. Their corresponding performance is evaluated later in chapter 4 and chapter 5. We then describe the implementation strategy for 3D wavefront applications in a single GPU system in section 3.3 which is evaluated later in chapter 6. We conclude this chapter with a discussion of various tuning trade-offs in section 3.4.

3.2 2D Wavefront Implementation

3.2.1 Single GPU Implementation and Tuning Points

The waveflow in figure 3.1 (a) illustrates the potential for parallel execution within wavefronts where the maximum potential parallelism occurs at the longest diagonal (in the figure the longest diagonal elements are shaded and numbered 4). In a naive implementation a data point can be evaluated as soon as its dependencies are satisfied. More pragmatically, when the problem size is large, it is a common optimization to partition the space into rectangular tiles, computing all points in a tile sequentially. The dependency pattern between tiles is identical to that between the original points, but with coarser granularity as shown in figure

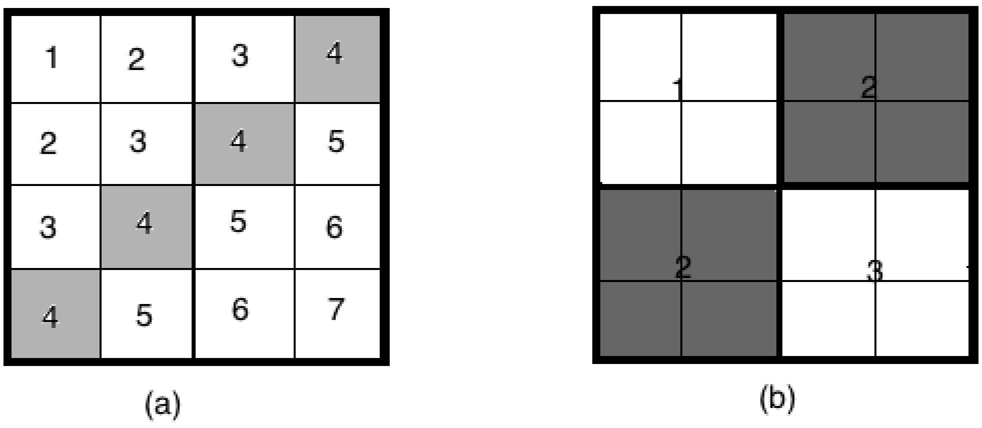


Figure 3.1: (a) Waveflow for a two dimensional instance of size 4 x 4 with maximum potential parallelism at 4th diagonal (b) The 4x4 grid now has 2x2 tiles with points inside the tile computed sequentially. Granularity is now coarser as two elements are now being computed in parallel, as opposed to 4, in the 2nd diagonal.

3.1 (b). Optimal selection of tile size is machine and problem dependent [60, 89]. For small problem instances, or very fine grained computations, it may even be the case that no parallelism can be usefully exploited. In such cases it can nevertheless be beneficial to tile the sequential evaluation order, to benefit from cache re-use.

Effective application of GPU acceleration is characterized by a need for regular computation (since the architectural building tiles of GPUs are Single Instruction Multiple Thread or SIMT (see section 2.2.3)), and sufficient granularity to amortize the costs of transferring data to and from the device and of initializing execution. In wavefront applications, there is clear scope for GPU parallelism across each successive diagonal, particularly since typical data point kernels are largely data-independent in structure, and hence SIMT-like collectively. Equally, it is intuitively clear that this will only be beneficial for diagonals of sufficient size and/or computational weight to outweigh the transfer overheads. This presents another machine and application dependent tuning point.

Our overall parallel implementation strategy therefore has three phases. In the first phase, tiled parallel computation proceeds using all cores of the CPU. In the second phase, execution switches to the GPU where it proceeds diagonal by diagonal. In the third phase, computation reverts to the CPU and is completed in tiled parallel fashion. The second phase, or in principle the first and third phases, may be null. This strategy is captured in our library code, using Threading Building Blocks [31] to control CPU phases and our own OpenCL harness to control communication with and execution upon the GPU.

As noted, within a diagonal, computation of each data point is independent, hence overall

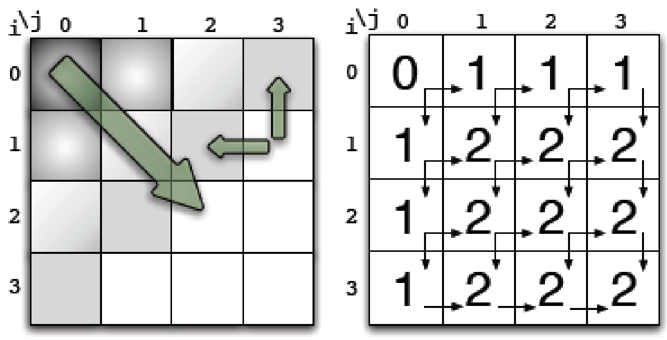


Figure 3.2: Dependency Array representation adopted from [28] for a 2D 4x4 wavefront. When the cell values of the dependency matrix decrease to zero, the requirement of the north and west elements having been computed is satisfied and threads are spawned.

diagonal computation is data parallel. The SIMT constraints of the GPU architecture are thus satisfied by the diagonal major representation of data and successive diagonals can be offloaded onto a GPU. For the remaining data points, CPU computation is preferable.

CPU threads are spawned for either each cell (no CPU tiling) or a block of cells (tiled CPU) in the data grid, only after their respective dependencies are satisfied. This can be seen in figure 3.2 which explains the dependency array representation of a 2D grid. Tiling within a GPU [1], reduces global memory access within the GPU and leads to local cache reuse, besides invoking fewer kernel calls from the host CPU. GPU tiles map to work-groups in OpenCL and the elements within the tile map to work-items or GPU threads. Within a work group, the work items have to be synchronized to follow the wavefront pattern. This introduces an overhead. The GPU tile size (our *'gpu-tile'*) tunable parameter is restricted by hardware and problem size.

Our single-GPU parallel implementation strategy therefore has three phases as previously noted, and three tunable parameters - number of diagonals to offload onto a GPU (or *'band'*) and the tile size of CPU and GPU(*'cpu-tile'* and *'gpu-tile'*). This implementation strategy is illustrated in the figure 3.3.

In our autotuning experiments for this scheme we first automatically determine whether to use parallelism, then autotune values for the tile size and for the number of diagonals to compute on the GPU, if any. The details of these tunings will be discussed in the later chapters.

3.2.2 Multiple GPU Implementation

The presence of multiple GPUs as explained in chapter 5 introduces two further tuning parameters. We must decide how many GPUs to exploit (tuning parameter *gpu-count*). Furthermore, partitioning data among multiple GPUs is non trivial and communication among

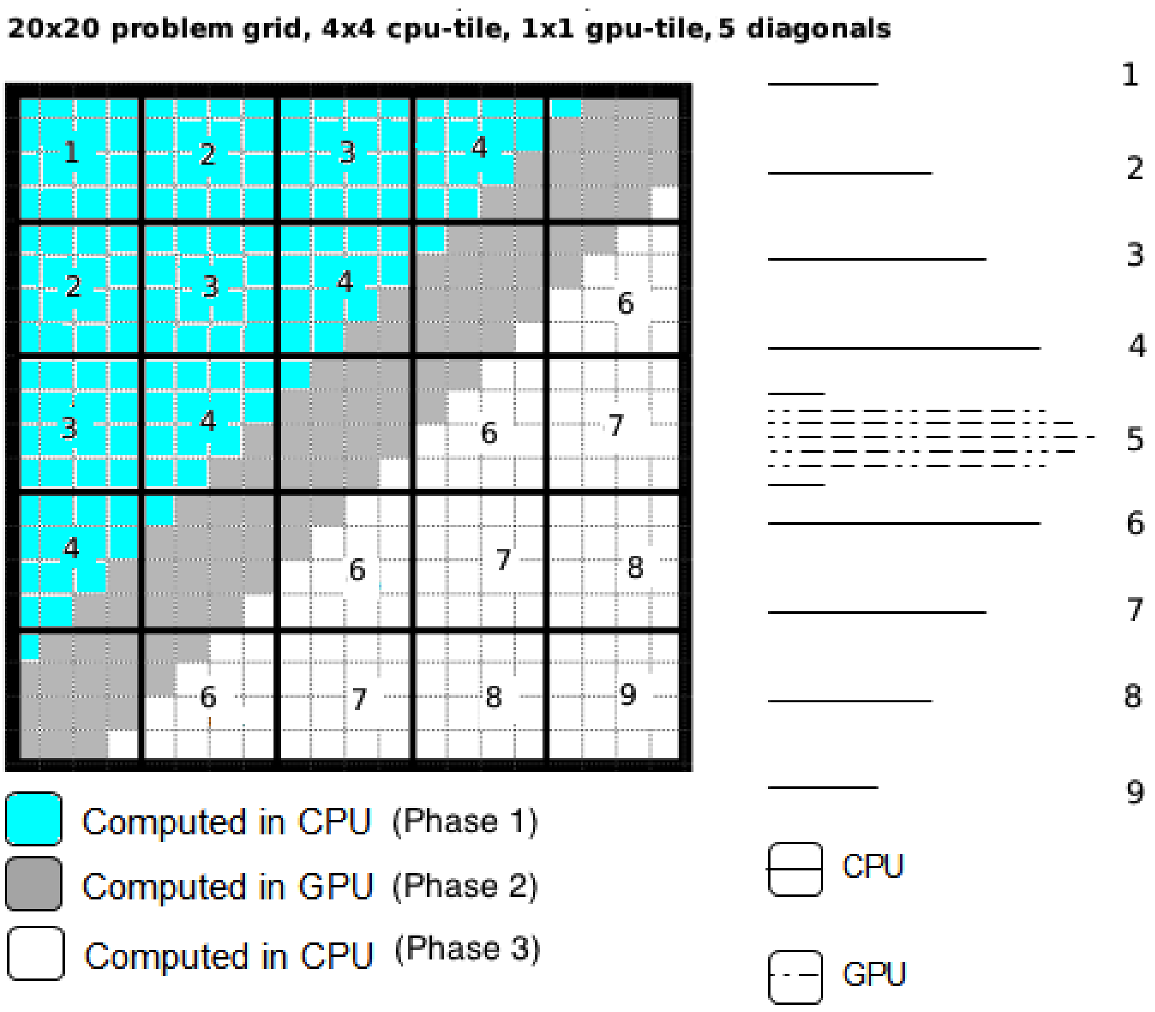

20x20 problem grid, 4x4 cpu-tile, 1x1 gpu-tile, 5 diagonals
1
2
3
4
5
6
7
8
9
Computed in CPU (Phase 1)
Computed in GPU (Phase 2)
Computed in CPU (Phase 3)
CPU
GPU

Figure 3.3: Implementation strategy showing three phase computation for 20 x 20 grid. Phase 1 and 3 have CPU tiles of size 4x4 and phase 2 is GPU consisting of its 1D work groups, with each kernel call corresponding to one diagonal. The straight and dashed lines to the right correspond to the number of concurrently computed elements in CPU and GPU respectively

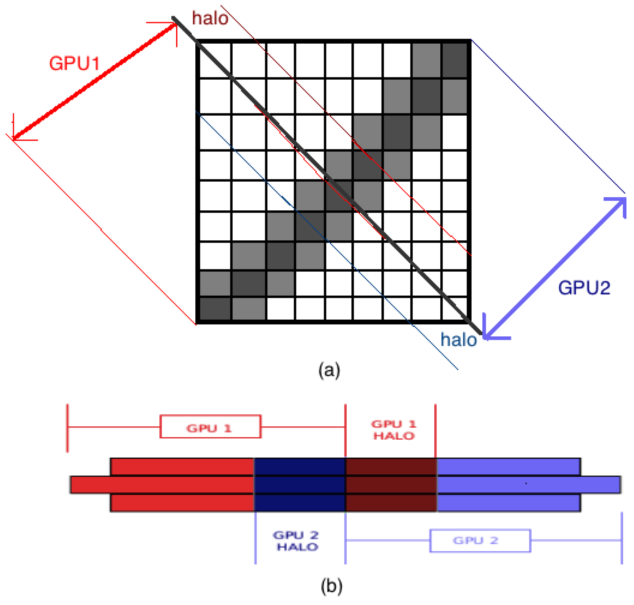


Figure 3.4: The partitioning of three diagonals among two GPUs with subsequent halo regions. The grid view in (a) shows partitioning of data among two GPUs with the three largest diagonals shaded. In (b) the three largest diagonals are represented as three horizontal bars

GPUs is expensive. Wavefront dependencies force data in the border regions (or *'halo'*) of partitioned diagonals to be shared among the GPUs. This is shown for two GPUs in figure 3.4. As successive partitioned diagonals within each GPU get computed, their border data becomes stale. This necessitates halo exchanges (or *'swaps'*) between the neighboring GPUs, depending on the extent of overlap or *halo* size. Each time this happens, data elements have to be first transferred to the host (CPU) memory and then transferred to respective destination GPUs. The overhead from data communication mandates minimizing communication between GPUs. However increasing *halo* size causes more redundant computation. Thus *halo* size is our fifth tunable parameter.

Table 3.1: Tunable Parameters

Parameter	**Description**
cpu-tile	side length of the square tiles for CPU tiling
band	number of diagonals on each side of the main diagonal, to be computed on the GPU
gpu-count	number of GPU devices to use
gpu-tile	the GPU equivalent of CPU tiling
halo	size of the halo for dual GPUs

To summarise, the tunable parameters in our implementation strategy are as listed in table 3.1. These will be the targets of our autotuning framework. In the next section we

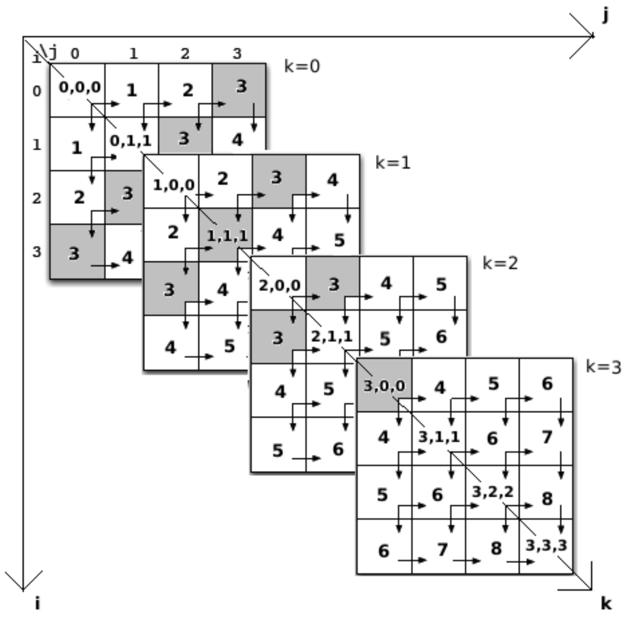


Figure 3.5: The planar waveflow. In a single sweep, all cells with coordinates $i+j+k=m$ lie on the m^{th} plane diagonal, and can be processed concurrently. Shown in grey are the elements lying in the 3rd plane diagonal, i.e. $i+j+k=3$

discuss tuning trade-offs for heterogeneous systems with multiple GPUs. The tunable three phase strategy itself is captured in our library code as stated earlier.

3.3 3D Wavefront Implementation

In this section we describe our implementation strategy for 3D wavefront applications. The waveflow for 3D computation grids contrasts with the diagonal traversal seen in 2D matrices. Now all elements that lie on a plane are computed simultaneously. Computation starts at position (0,0,0) and propagates to neighboring elements in a series of diagonal planes. For simpler analysis, we consider cubic datagrids of size $n \times n \times n$. This results in planar wave flows for a cube of dimensions (i, j, k) with all elements lying on the m^{th} plane having coordinates $i+j+k=m$. An example of such simultaneous computation on plane 3 is shown in figure 3.5.

Similarly, the dependency array for a 2D wavefront becomes a 3D dependency array. Figure 3.6 shows the dependency arrays associated with the planar waveflow of a 3D wavefront. As observed, in most cases computation of an element depends on its north, west and top elements. This leads to the following dependency relationship.

- $\forall x_{i,j,k}, (i > 0, j > 0, k > 0)$, dependency $= 3$
- $\forall x_{i,j,k}, (i > 0, j = 0, k > 0) \vee (i = 0, j > 0, k > 0)$, dependency=2

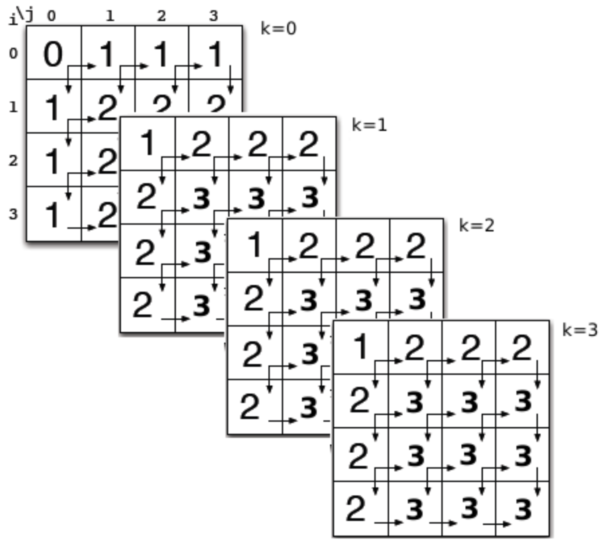


Figure 3.6: The dependency graph for the planar flow. Each cell depends on the left, right and top elements. Once the dependency decreases to 0 from a maximum of 3, the cell is processed.

- $\forall x_{i,j,k}, (i=0, j=0, k>0)$, dependency=1

Taking these into consideration we discuss our implementation strategy for the 3D case. Due to the addition of another dimension, the GPU-CPU partition strategy has to be modified. More specifically, the definition of *band* used previously in table 3.1 has to be modified to refer to planar diagonals. The longest diagonal of a 2D matrix consists of elements with coordinates $i+j=m+1$ for an $m*m$ square matrix. In a 3D grid the longest planar diagonal consists of elements with coordinates $i+j+k=n$ where n is the maximum number of elements that can be computed in parallel. For an $m*m*m$ cubic data grid, the values of n for even and odd cases are shown in Equation 3.1 and Equation 3.2 respectively.

$$n = 3*(m-1)*0.5+1 \quad m : \exists k \in N, m = 2k \tag{3.1}$$

$$n = 3*(m-1)*0.5 \quad m : \exists k \in N, m = 2k+1 \tag{3.2}$$

Like the 2D wavefront, we linearize the successive planar diagonals to be offloaded onto the GPU. These are then computed in a SIMT diagonal major fashion, with each kernel call corresponding to a plane diagonal of concurrently computed elements. However, unlike the 2D case where offset computation for north and west elements was simple, here computing the position of north, west and top elements is quite complex. This can be seen from figure 3.7 which shows the linearized versions of the 3D and 2D cases. In the 2D case, there is regular increase in number of elements until the main diagonal and after that there is a regular decrease. This index computation for north and west elements is computed taking into account the offset values for positions on or before and after the main diagonal.

In the 3D case offset calculation is complex, requiring a computation that checks different conditions that include the following :

- Diagonal plane position, i.e., is it after or before the main plane diagonal (as observed for diagonal plane 3 in Figure 3.7 (a));
- Number of elements in the current diagonal plane, as even and odd number of elements require different offset calculation strategies;
- Whether the current diagonal plane is the main plane diagonal (as observed in Figure 3.7 (b))
- Number of elements in previous diagonal plane. In the 2D case this was not needed because there is a simple unit increase in number of elements until the main diagonal, followed by unit decrease post main diagonal.

The SIMT constraints of a GPU for multiple if-else conditions (see subsection 2.2.3) makes such an algorithm costly. Thus for index computation of north, west and top elements, we have chosen to use an additional helper array containing the (i, j ,k) coordinates of each element from the linear diagonal major array which was offloaded onto the GPU. Since the transfer of this auxiliary array is done only once, the overhead of additional data is easily offset from the gains made by avoiding the complex computation that would be needed for index calculation inside the GPU. The performance benefits of this strategy are evaluated in subsection 6.4.1.

3.4 Performance Tuning Trade-Offs

Parallel Computing becomes feasible when there is enough parallelism to be exploited. This is especially true for the wavefront where computation starts from a corner with only one element that can be computed to many elements in the center diagonals and back to one element in the opposite corner. This diamond shaped number of parallel elements means it is vital to select the diagonals with enough parallel elements that can be offloaded onto an accelerator like the GPU to overcome communication costs while the remaining elements are computed in CPU. In the diagonal major representation of the computation grid, the elements of the first diagonal offloaded onto the GPU belong to the *start diagonal* and the last diagonal before computation switches to CPU is the *end diagonal.* This is shown in Figure 3.8.

Thus there are two important considerations for going parallel in a wavefront pattern.

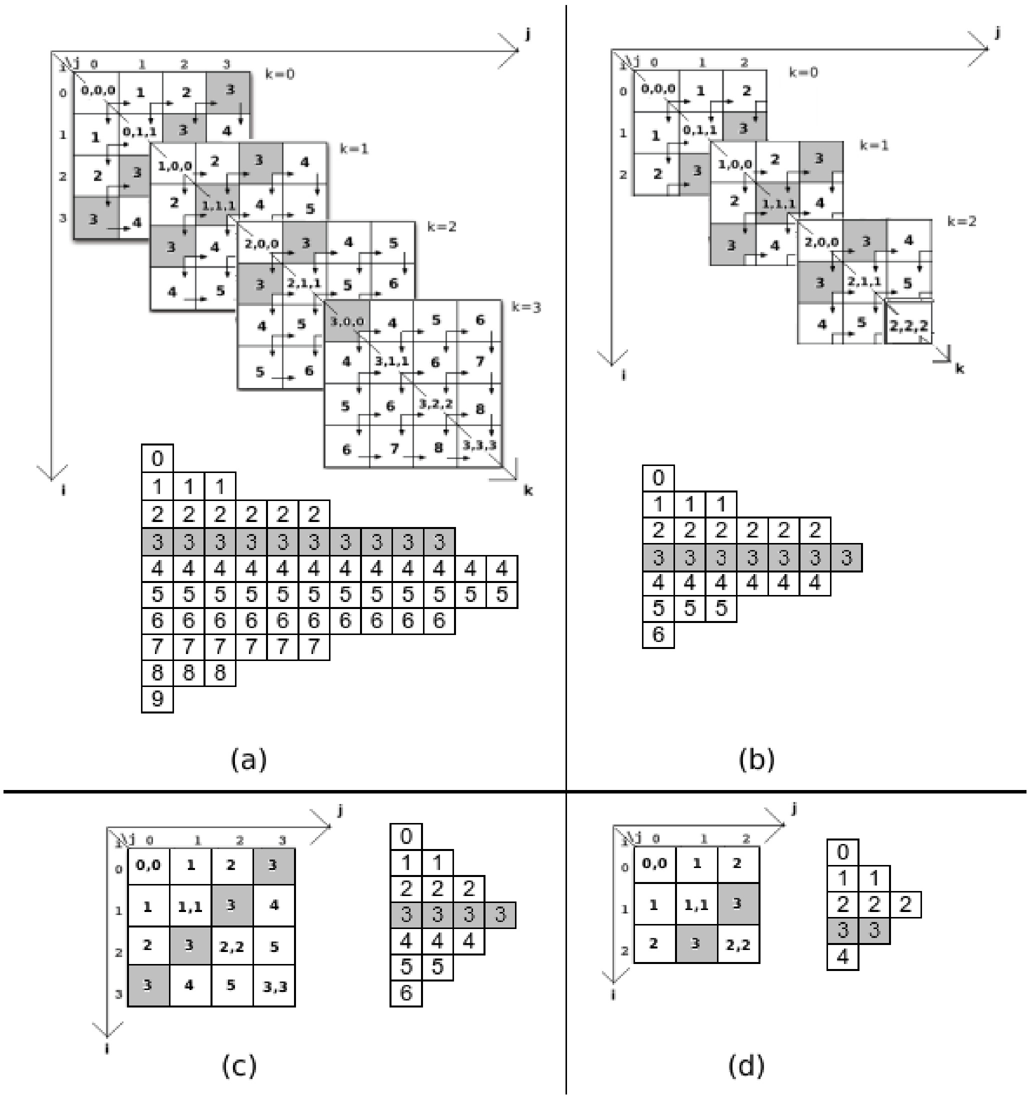


Figure 3.7: The 3D compute grids of size $4 \times 4 \times 4$ featuring even number of elements and of size $3 \times 3 \times 3$ featuring odd number of elements are linearized into diagonal major arrays in (a) and (b) respectively. Their shapes make index computation complex and costly inside the GPU. This contrasts with the regular shapes of linearized diagonal major representation for the 2D compute grids of size (c) 4×4 and (d) 3×3 grids which makes index computation simple.

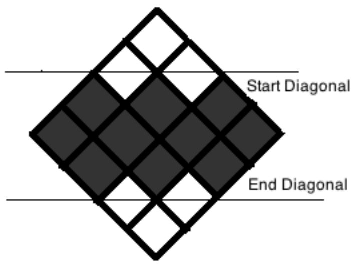


Figure 3.8: Diagonal major representation of a 2D wavefront, with all elements between the Start and End Diagonals offloaded onto GPU (shown in grey). Remaining elements are computed on the CPU.

- The problem size (*dim*) should be large enough, since smaller sized problems can be computed quicker in the faster CPU cores
- The granularity of task (*tsize*) should be high enough so that computation dominates over the cost of starting a GPU and the communication overhead of transferring data between GPU and CPU.

This communication cost naturally increases when data size (*dsize*) being transferred increases. Another factor that increases communication cost is the number of GPUs employed. While with a single GPU data is transferred from/to CPU only twice, dual GPUs have the additional overhead of exchanging neighboring data between themselves every few iterations (*halo* swapping). This overhead becomes more expensive if the data size is large as more time is spent in swapping halos. A reduction in halo swaps is obtained by increasing the *halo* size.

The diagonal major structure of the problem grid in the GPU restricts this *halo* size to a maximum of the length of the start/end diagonal. Even at maximum size, the advantage gained from fewer swaps has to be traded against redundant computation, which starts affecting performance with increasing granularity of task.

Communication cost is also affected by tiling (*gpu-tile*) the GPU since this reduces the number of kernel calls required but incurs the additional cost of synchronizing work items within each work group. If computation dominates over communication anyway, time spent in kernel calls no longer matters and tiling would then prove to be counter productive.

System characteristics affect performance: a fast GPU coupled to a slow CPU means data will mostly be offloaded to the GPU (unless bandwidth is the bottleneck) leading to higher values of *band.* In such a system, CPU tiling will have negligible effect as most of computation is carried out in the GPU. Likewise, in fast CPU-fast GPU systems, good *band* values will be correspondingly lower.

3.5 Summary

In this chapter we looked at the implementation details of our framework. We first discussed the single GPU implementation then the multi-GPU implementation. Then we discussed the enhancements to the 2D wavefront tuning framework to support a 3D wavefront. All these implementations are transparent to the user and auto-tuning takes place offline based on profiling specific to the architecture and application. We concluded this chapter with an overview of the various tuning trade-offs that are expected to affect overall execution speed of the applications.

Chapter 4

Single GPU Autotuning : 2D Wavefronts

4.1 Introduction

In this chapter we discuss the autotuning strategy for 2D wavefront applications and analyze the results from our exhaustive search and machine learning models.

We begin with a discussion of the tuning methodology for single GPU heterogeneous systems followed by a brief overview of the 2D wavefront applications deployed on our systems in section 4.2. We then analyze the results of our autotuner for those applications by comparing them against the best performing configurations found from an exhaustive search in subsection 4.3.1. We repeat this process across three contrasting heterogeneous systems in subsection 4.3.3 - subsection 4.3.5. We summarize our findings in section 4.4.

4.2 Autotuning Strategy

Our goals are to understand the relationship between settings of the internally tunable implementation parameters (sequential or parallel, tiling, CPU-GPU partitioning) and performance, and to use machine learning techniques to control the automatic setting of these parameters.

To address the first goal, we conduct an exhaustive evaluation of a problem space covering our three applications, across a range of settings for four key parameters. *dim* and *tsize* represent the problem size and kernel task granularity respectively. For simplicity we assume "square" problems, so *dim* is a single integer. The underlying unit of granularity is the time taken to execute a single instance of the kernel function on a single CPU core. However since this differs by many orders of magnitude across our problem set, we choose, for presentational purposes, to report on *tsize* in terms of an application specific normalized multiple of this

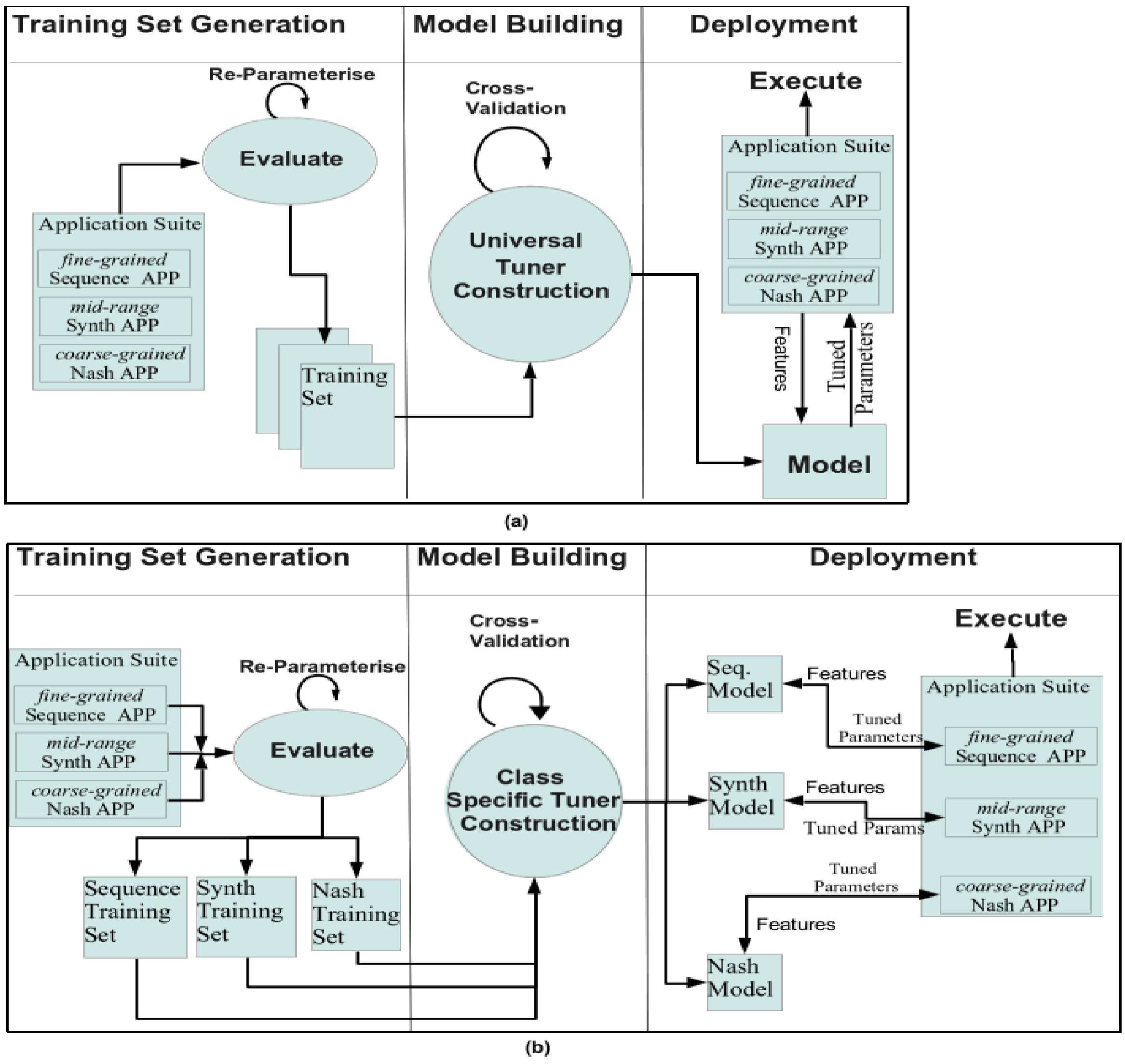


Figure 4.1: (a)Tuning Strategy for Universal Tuner (b) Tuning Strategy for Class Specific Tuners

underlying value. We stress that this normalization is not present in the raw data used for machine learning. *dim* and *tsize* are inputs to the tuning heuristic. *tile* and *band* describe the tunable parameters. As discussed earlier in subsection 3.2.1, *tile* is the side length of the square tiles for CPU tiling and *band* captures the number of diagonals, on each side of the main diagonal, to be computed on the GPU. Thus a *band* of n means that $2n+1$ diagonals in total are assigned to the GPU (so a band of -1 means that the GPU is not to be used). To address the second goal, we first build a binary SVM based predictor to decide whether or not to exploit parallelism. For those cases in which parallelism is predicted to be beneficial we then apply and evaluate two machine learning heuristics, based on Linear Regression and SVM regression respectively [37]. Background information relating to these techniques has been provided in subsection 2.4.4 and subsection 2.4.5.

In our initial auto-tuning experiments we derived a single model for each learning technique, using training data drawn from all three applications. We refer to these models as *universal* tuners as shown in Figure 4.1(a). Subsequently we experimented with models derived from training sets drawn exclusively from each application in turn. We refer to these as *class-specific*, as tuners shown in Figure 4.1(b). These tuners are interesting because they incorporate prior domain knowledge about the application and should perform better than the *universal* tuners. We compare the performance of these two approaches. Of course, to apply the class-specific approach in a deployed system would involve a further application classification phase.

In all cases, training sets are created by sub-setting the exhaustive search space as follows: firstly a subset of the problem instances (i.e., by *dim* and *tsize*) are selected by regular sampling. This is to ensure the training examples are separate from the evaluation examples. Then, the best five performance points for these instances (by *tile*, *band* and sequential/-parallel decision values) are added to the training set. Our choice of best five is a heuristic based on empirical observation of the prediction accuracy of our learners for the number of training samples. The intuition is that these should be representative of the good decisions we wish to embed in our models. We repeat this procedure independently for each system, in line with a scenario which would see the software trained "in the factory".

To assess the quality of our learned models, we apply them to a range of problem instances which were not in the training set, as would happen after deployment in a real scenario. We compare the performance obtained by our heuristically generated tunings with the best performance found for the same problem instances during exhaustive search.

4.2.1 Application Suite and Platforms

4.2.1.1 Applications

We have conducted our experiments with three wavefront applications, and a set of instances distinguished by an adjustable internal parameter for two of these applications, which impacts upon the granularity of the kernel. By varying this parameter we can explore a very wide range of wavefront instances. This is also beneficial during machine learning, easing generation of synthetic training data. Two of our applications, the Nash Equilibrium and Biological Sequence Comparison have already been discussed in detail in section 2.6. We provide a recap of these two applications followed by an explanation of the synthetic application.

Nash Equilibrium [104] : a game-theoretic problem in Economics, characterized by small instances but a very computationally demanding kernel. The internal granularity parameter controls the iteration count of a nested loop.

Biological Sequence Comparison [95] : a string alignment problem from Bioinformatics, characterized by very large instances and very fine-grained kernels, varying with detailed comparisons made. This application does not have an internal granularity parameter.

Synthetic application : Noting the contrast in typical problem size and granularity between our first two applications, we created a third, synthetic application to facilitate experiments in the "mid-range". The code is based on the Gauss-Seidel iterative solver which is a technique for solving linear system of equations of the form $A\mathbf{x} = \mathbf{b}$. The equations are solved one at a time in sequence, using previously computed results as soon as they are available. It is defined by the iteration

$$L\mathbf{x}^{(k+1)} = \mathbf{b} - U\mathbf{x}^{(k)} \tag{4.1}$$

where L is the lower triangular matrix and U is the strictly upper triangular matrix(see section 2.6.3). The solution for equation 4.1 by forward substitution is given by

$$x_i^{(k)} = (b_i - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)})/(a_{ii}) \tag{4.2}$$

Our synthetic application is derived from this iterative solver, and carries out computation in a wavefront pattern based on data from the *north*, *west* and *north-west* elements. The synthetic application allows us to simulate a range of real world applications whose task granularity lies between that of the fine grained Sequence Comparison and the coarse grained Nash Equilibrium application. The granularity of the tasks is controlled by an internal iterator which carries out per element computations repeatedly to simulate coarse grained computations. The synthetic application thus provides useful training data for our autotuner by simulating mid-range wavefront applications. It is a strength of the pattern-oriented approach that such an approach is feasible, removing the need to find real mid-range granularity applications for the training phase.

4.2.1.2 Systems

Our experiments were conducted across the three systems described in table 4.1. These contrasting systems were selected to be representative of the range of heterogeneous systems available today. The i3-540 is an example of a slow CPU with 1.2 Ghz clock speed, coupled to a relatively fast GPU GTX 480. The i7-990 has a faster GPU, GTX 580, and fast i7 CPU cores with 1.6 GHz. It has the highest number of CPU cores and memory size among our systems. The i7-3280 has fewer cores than the i7-990 but has the fastest clock speed at 3.6 GHz and a fast GPU, the HD 7970. Contrasting the two i7 systems, computation would be faster on the i7-3280 while more data can be loaded onto the i7-990.

We measure runtime of the whole program execution using wall clock timers in the host program, averaging across three runs. We observed extremely low variance in the region of few hundred milliseconds.

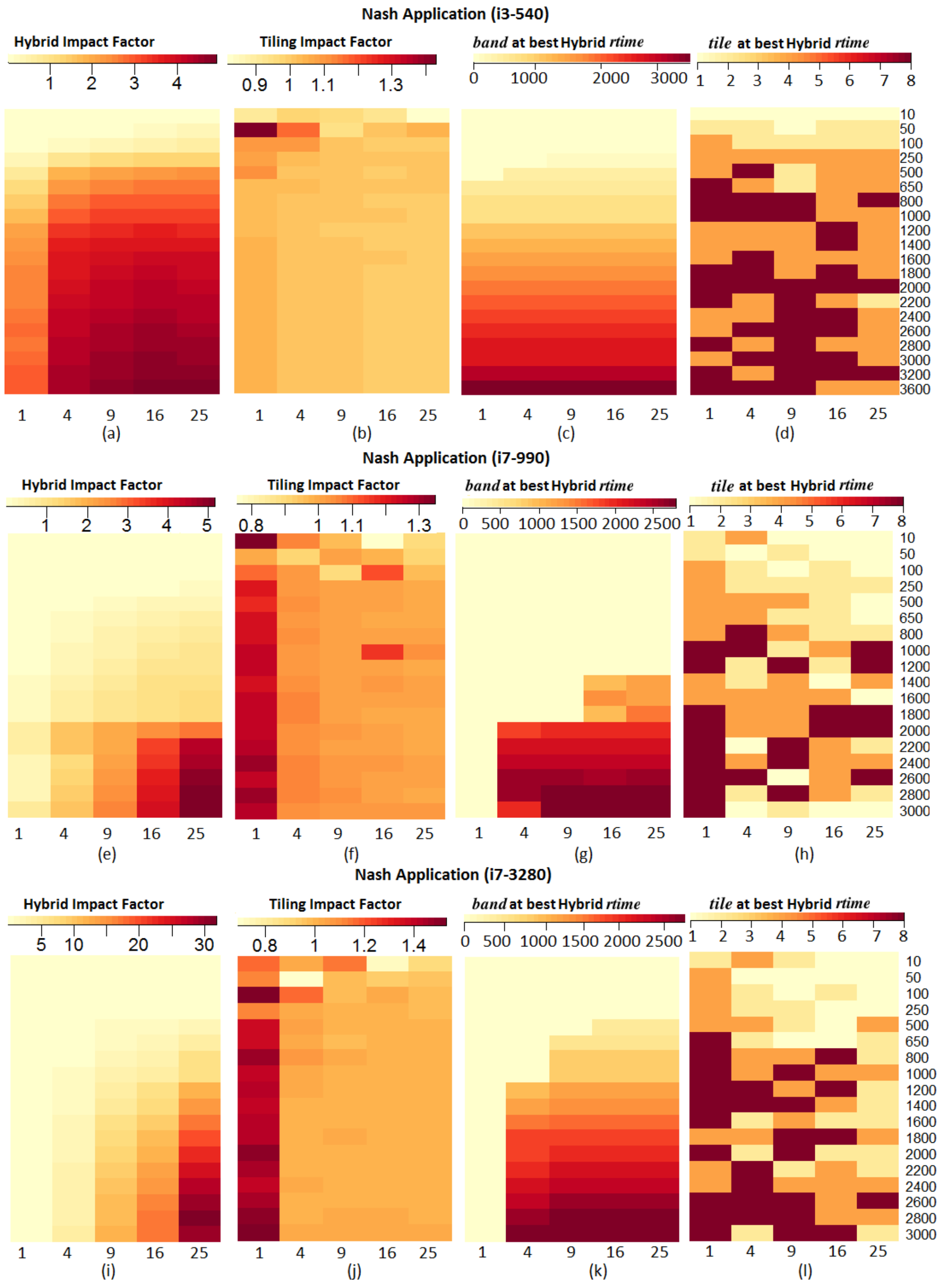


Figure 4.2: Exhaustive search results for the Nash application on three systems. Maps illustrate the potential performance speed-up factor of the best hybrid solution against the best parallel CPU only solution, the same for tiled against non-tiled solutions, and the *band* and *tile* values at the best points. In all maps the x-axis is *tsize*, indicating kernel task granularity and the y-axis is *dim*, indicating problem size.

Table 4.1: Experimental Systems

System	CPU Speed (Ghz)	Cores (HT)	Mem (GB)	GPU	Speed (Mhz)	Mem (GB)
i3-540	1.2	4	4	GTX 480	700	1.6
i7-990	1.6	12	12	GTX 580	772	1.6
i7-3280	3.6	8	8	HD 7970	925	3.0

4.3 Results and Analysis

We present the results of our exhaustive search of the problems spaces across all applications and architectures in subsection 4.3.1. Then, in subsection 4.3.2, we investigate our autotuning strategies followed by detailed inspection of autotuning results for nash application, synthetic application and sequence comparison application in subsection 4.3.3, subsection 4.3.4 and subsection 4.3.5 respectively.

4.3.1 Exhaustive Search Results

Nash Application

We first investigate the Nash application results for all three systems. Figure 4.2 presents a set of four performance maps for each system, with all maps having *tsize* and *dim* as axes. In each set of four, the first map compares runtime (*rtime*) of the best tiled parallel CPU-only setting with performance on the best tiled and banded parallel CPU+GPU combination ("Hybrid" in the figure). This is expressed as the ratio of the best tiled parallel CPU-only *rtime* to that of the CPU+GPU combination, so values < 1 indicate the CPU-only version being faster, while values > 1 indicate the potential benefits of correct *band* tuning. The second map captures the impact of *tile* selection which is similarly expressed as the ratio of best untiled (i.e. 1x1 tile) *rtime* to best tiled (i.e $>$ 1x1) *rtime*. Thus, values > 1 indicate that tile size of more than 1x1 led to better *rtime*. The third and fourth maps for each system respectively indicate the *band* and *tile* values at the best CPU-GPU combination. Inspection of this data allows us to understand interesting characteristics of the problem space.

For the i3-540, offloading almost all of computation onto the GPU from *dim*=500 onwards leads to improvement in *rtime* as seen from 4.2(a). The *band* value is uniform across all values of *tsize*, meaning it is unaffected by granularity. CPU tiling offers little benefit, as the GPU dominates and most of the computation is offloaded onto the GPU.

Results for the i7-990 show that the *dim* at which to start using the GPU comes later (at *dim*=1400 in 4.2(e) vs *dim*=250 in 4.2(a) for *tsize*=25). Fractional values for *rtime* in 4.2(e) until *dim*=1200 for *tsize*=[4, 9, 16, 25] and the corresponding *band*=-1 (CPU only) in 4.2(f) show the faster CPU cores of the i7-990 dominate performance until problem sizes are

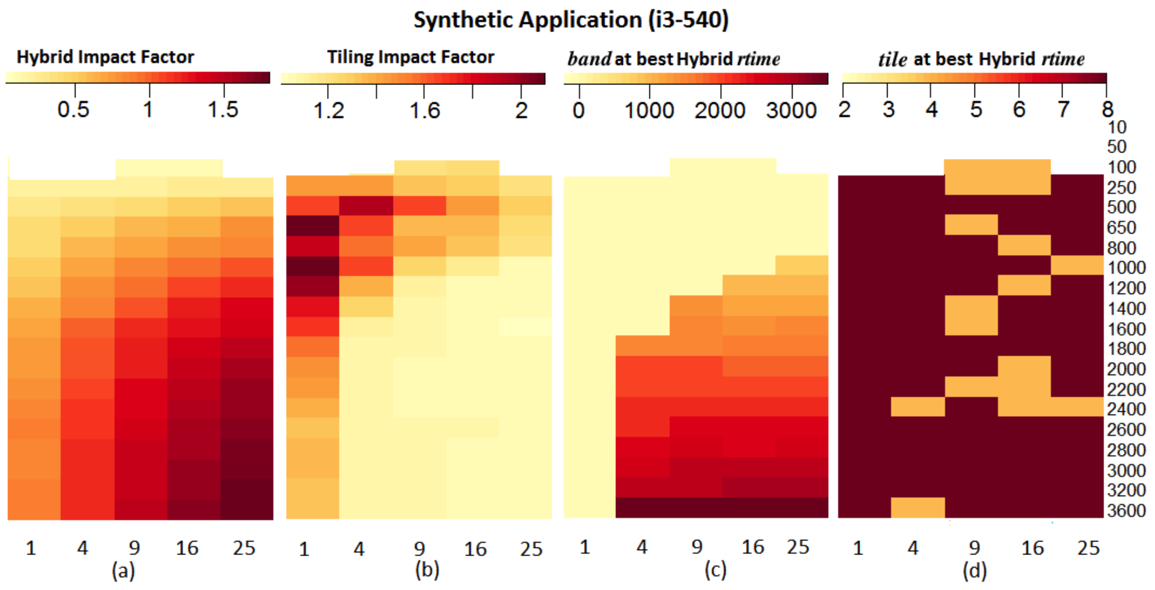


Figure 4.3: Exhaustive search results for the synthetic application on the i3-540. Maps illustrate the potential performance speed-up factor of the best hybrid solution against the best parallel CPU only solution, the same for tiled against non-tiled solutions, and the *band* and *tile* values at the best points. In all maps the x-axis is *tsize*, indicating kernel task granularity and the y-axis is *dim*, indicating problem size.

sufficiently large to offload computation onto the GPU. Tiling is vital for cache reutilization when *tsize*=1 as seen in 4.2(f), unlike the i3-540 in 4.2(b), but *tile* values vary arbitrarily from 2 to 8 as seen in 4.2(h).

For the i7-3280 the effect of the raw GPU performance advantage is apparent from the 30 times peak *rtime* speedup potential against 5 times speedup for the i3-540. The fast CPU cores mean GPU benefits appear later (at *dim*=650 in 4.2(i) for *tsize*>16). Tiling is crucial when most computation is carried out in its fast CPU cores as seen for *tsize*=1 in 4.2(j).

Synthetic Application

For the synthetic application, a tiled sequential implementation was found to be best in (i) *dim*=[10,50], all *tsize* values (ii) *dim*=100, *tsize*=[1,4,25]. For parallel cases, the results of i3-540 are shown in figure 4.3. Tiling is more effective than offloading computation onto the GPU in cases where *tsize* and *dim* is small. For *tsize*=1, the CPU cores profit from tiling as seen from a peak of more than twice the speedup in *rtime* in 4.3(b). When *tsize* is increased, offloading computation to the GPU is beneficial. Unlike Nash, *tile* size of 8x8 was usually best.

For the i7-3280, *band* value at the best *rtime* was consistently -1 as illustrated in Figure 4.4 (c), meaning for best performance all computation should be carried out in the CPU cores without offloading onto the GPU. Since all computation was carried out in the CPU, the speedup from parallel CPU only phase (phase 1 of hybrid computing using Threading Building Blocks, refer to subsection 3.2.1) over serial computation is shown in Figure 4.4(a).

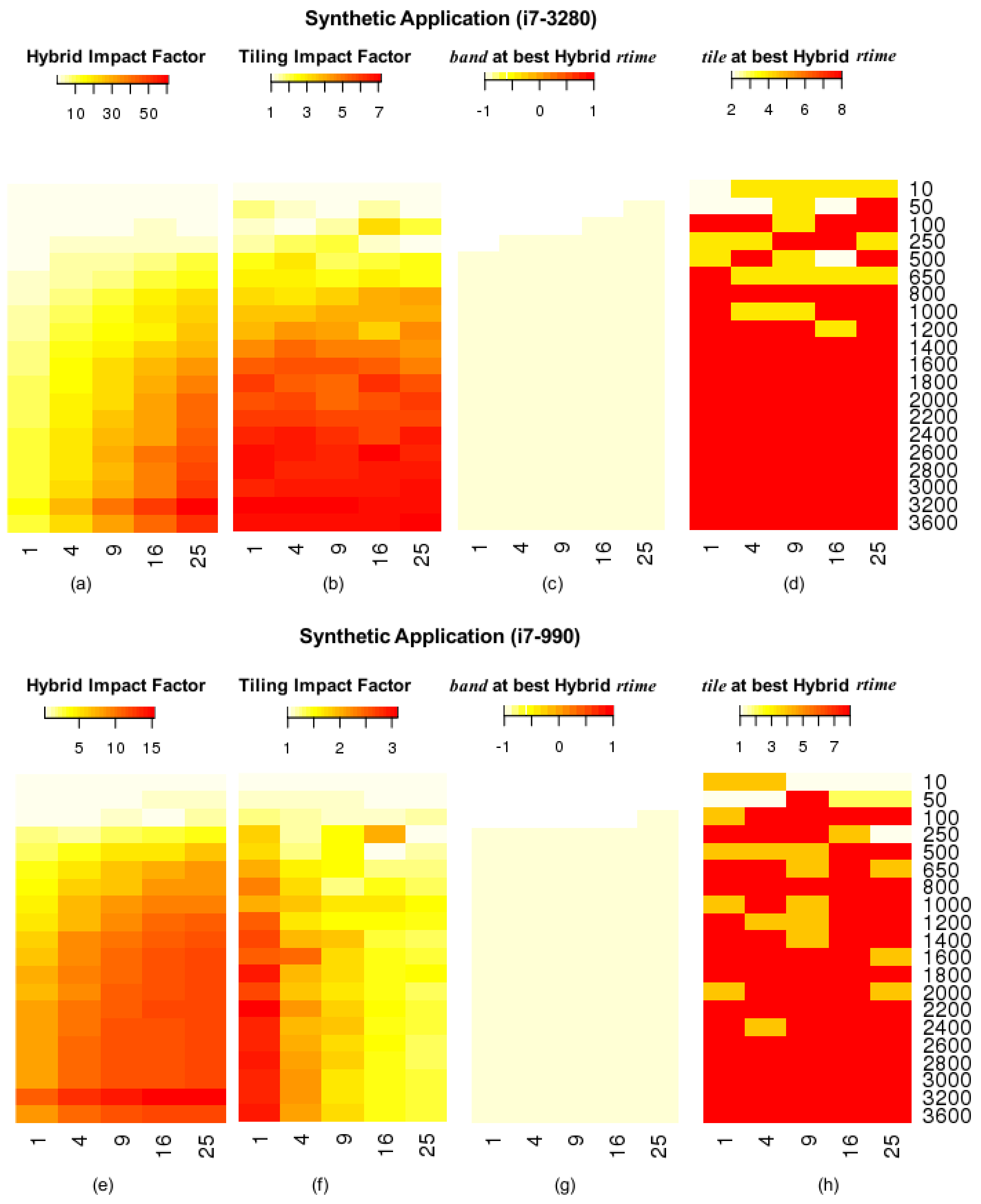


Figure 4.4: Exhaustive search results for the synthetic application on the i7-3280 and i7-990 systems. Since the best points are CPU only and the GPU plays no role, here maps illustrate the potential performance speed-up factor of the best parallel CPU only solution against the serial solution, the same for tiled against non-tiled solutions, and the *band* and *tile* values at the best points. In all maps the x-axis is *tsize*, indicating kernel task granularity and the y-axis is *dim*, indicating problem size.

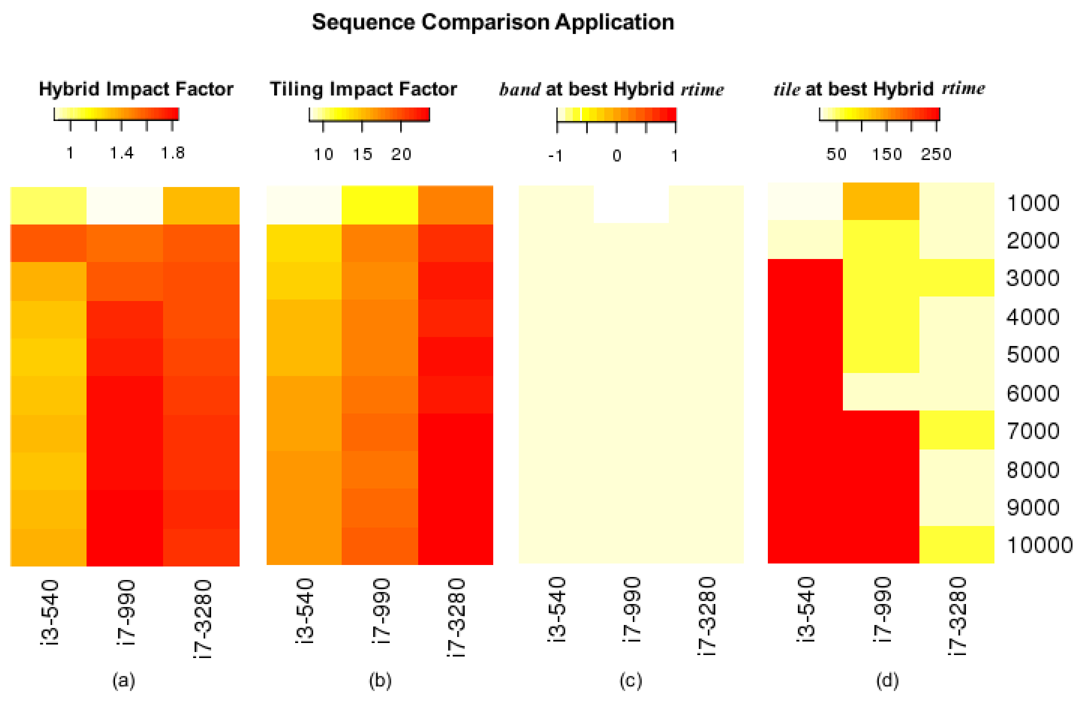


Figure 4.5: Exhaustive search results for the Sequence Comparison application on the i3-540, i7-990 and i7-3280 systems. Since the best points are CPU only with the GPU playing no role, here the maps illustrate the potential performance speed-up factor of the best parallel CPU only solution against the serial solution, the same for tiled against non-tiled solutions, and the *band* and *tile* values at the best points. In all maps the x-axis is system (i3, i7-990 and i7-3280), and the y-axis is *dim*, indicating problem size. It should be noted that the sequence comparison application has constant *tsize*, hence there is no variation for this parameter.

The *tile* sizes were also generally 8x8 for *dim*>1200 across all *tsize* values, as observed from Figure 4.4(d). Similar *band* and *tile* results were also obtained for the i7-990 as illustrated in Figure 4.4 (g), (h).

The fast i7 CPU systems are affected by tiling with *tile* sizes > 1 leading to almost 7 times higher speedup in the i7-3280 for all instances from *dim*>2400 across all *tsize* values. Tiling is less effective in the i7-990 which possesses a slower CPU compared to the i7-3280. The tiling impact factor in both systems contrasts with the i3 system where it is < 1.2 for all configurations of *dim*-*tsize* when *band*>-1. This is explained by the dominance of the relatively fast GPU versus slow CPU in the i3 system, where it makes sense to offload computation onto the GPU, rendering CPU tiling irrelevant in those parts of the optimization space.

We also note the tiling impact factor of the i7 systems for this “mid-range” application is larger than the same for the coarse grained Nash application, as illustrated in Figure 4.2 (f) and (j). This is because computation is dominated by communication overheads resulting in *band* values of -1 with tiling affecting *rtime*, while in Nash application the reverse is true for most parts of the optimization space.

Sequence Comparison Application

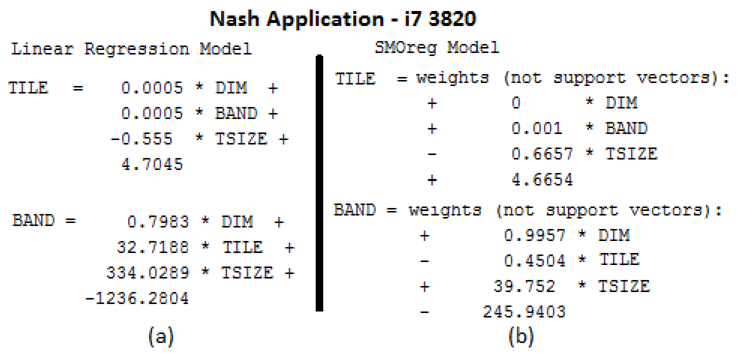
For Sequence Comparison, the best results were usually obtained by the tiled CPU-only parallelism, except for the i7-990 where the tiled serial version in *dim*=1000 was typically a few percent faster due to spawning overhead as illustrated in Figure 4.5. Since *band*=-1 across all systems, the hybrid impact factor is measured as the speedup of parallel CPU phase over sequential phase, similar to the synthetic application. This hybrid impact factor is however, quite low with a maximum value of 1.8, compared to being almost 30 for the synthetic application. This is due the low task granularity of the sequence comparison application which makes spawning new threads expensive.

4.3.2 Autotuning Experiments

Our autotuning experiments are carried out using two learning models - Linear Regression and Support Vector Machines (SVM). Linear Regression is the simplest predictive model with a squared error score function, and is fast to train. So this was our first choice to investigate the relationship of our tunable *band* and *tile* parameters and input features. Then we investigate the tuning performance from using SVM which can partition the search space on higher dimensions and provide potentially better results.

Our first tuning decision is whether or not to exploit parallelism at all. The Binary SVM predictor was a universal tuner with an average accuracy of 97% across all systems during cross validation on the training set. For our test set it correctly predicted in all cases except for *dim*=[700,900] in the Sequence Comparison application, in i7-990.

For parallel implementations the exhaustive search of the optimization space highlighted the need for choosing optimal values for *band* and *tile* parameters. Since we may have dependence between these parameters, we investigate two models, Linear Regression and SVM Regression (using a polynomial kernel with a linear exponent) to tune our output parameters. Figure 4.6 gives an example of the type of models generated by these methods, illustrating the predicted dependence between *band* and *tile* values.



Nash Application - i7 3820

```
Linear Regression Model

TILE   =     0.0005 * DIM   +
             0.0005 * BAND +
            -0.555   * TSIZE +
             4.7045

BAND =       0.7983 * DIM    +
            32.7188 * TILE   +
           334.0289 * TSIZE +
         -1236.2804
```
(a)

```
SMOreg Model

TILE   = weights (not support vectors):
        +        0       * DIM
        +        0.001   * BAND
        -        0.6657 * TSIZE
        +        4.6654

BAND = weights (not support vectors):
       +        0.9957 * DIM
       -        0.4504 * TILE
       +       39.752   * TSIZE
       -      245.9403
```
(b)

Figure 4.6: Sample models of *tile* and *band* relationships as predicted by (a) Linear Regression (b) SVM Regression.

The key advantage of autotuning over exhaustively searching for best performing points is the significant reduction in tuning time. While it took a long time to collect exhaustive search results which were in the order of few days, it took only a few hours to train our learning models. Once the learners were trained, validating against a test set was even quicker, in the magnitude of tens of minutes. Finally, tuning decisions for new wavefront applications (using trained learners) were made instantaneously.

In the next subsections we will illustrate autotuning performance through a series of heat maps. For each system we have a pair of maps, one for Linear Regression and one for SVM regression. In each pair, the first map compares the performance of our tuned implementation with that of the best implementation found during the exhaustive search for the same problem instance. This is expressed as the improvement in performance from autotuned to best exhaustive as percentage of runtime of the best exhaustive runtime (abbreviated to *ber* below). Thus, positive values indicate that autotuning outperformed exhaustive search. This is possible because the tuner may select *band* or *tile* values which were not tried during exhaustive search. The second map in each pair indicates the percentage difference between autotuned and best exhaustive *band* values at the same points.

4.3.3 Autotuning Results for Nash Application

Autotuning results for the Nash application on each of our three systems are presented in Figure 4.7, Figure 4.8, and Figure 4.9.

Figures 4.7(a) and 4.7(b) show results for the i3-540 by the universal tuner, which performed as well as the class specific tuner. Both Linear and SVM Regression models predicted values close to optimal values. The class specific tuner fares better than the universal one as observed from higher *rtime* improvement, up to a maximum of 60% in 4.7(c) and up to 30% in 4.7(d). Here the SVM tuner performed better on average because it was accurate over most *dim-tsize* values. This is observed from 4.7(d) having a higher number of points with 0% difference between best *band* and predicted *band* values in comparison to 4.7(c). The *rtime* values are slightly different from exhaustive search due to difference in predicted *tile* values.

The universal tuner fared poorly for the i7-3280, with predicted values increasing *rtime* by an average of 120% against *ber*. This is because in the faster i7 CPU systems, the best *band* values are generally -1. In relative terms, the Linear Regression based tuner fared better than the Support Vector one. This is illustrated from heatmaps in figures 4.8(a) and 4.8(b). In 4.8 (a) the Linear Regression tuned performance is almost 300% worse at points where the predicted *band* values are more than 50% apart from the best *band* values. However in 4.8(b) the SVM regression tuned performance is nearly 3000% worse at some points where *band* values differ by more than 60%. The *rtime* is affected by *band* values more

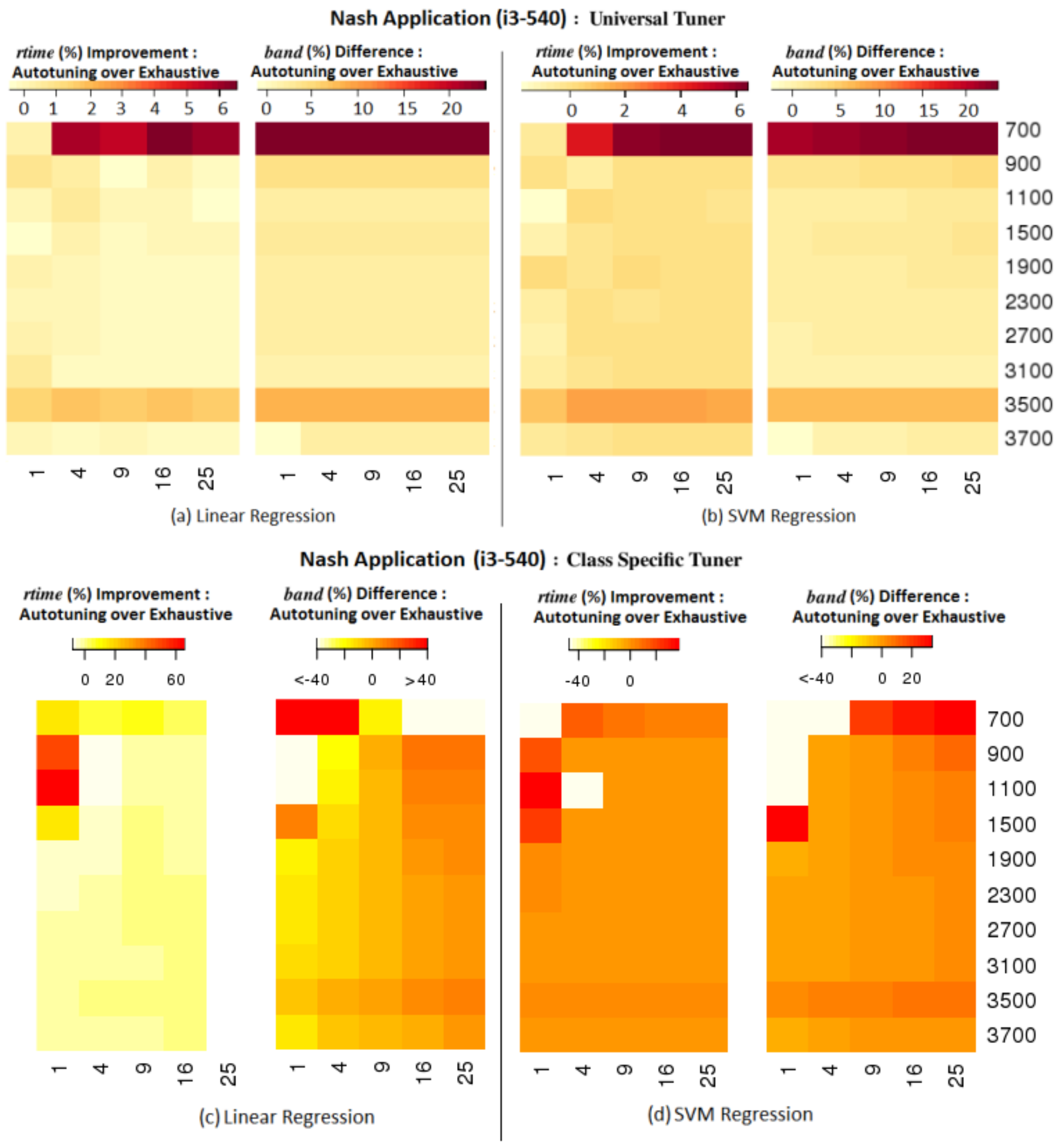


Figure 4.7: Autotuner performance for the Nash application on i3-540 for class-specific tuners. Each pair of maps shows autotuned performance against the best performance from exhaustive search, and the difference in *band* values used at these points. In all maps the x-axis is *tsize* and the y-axis is *dim*.

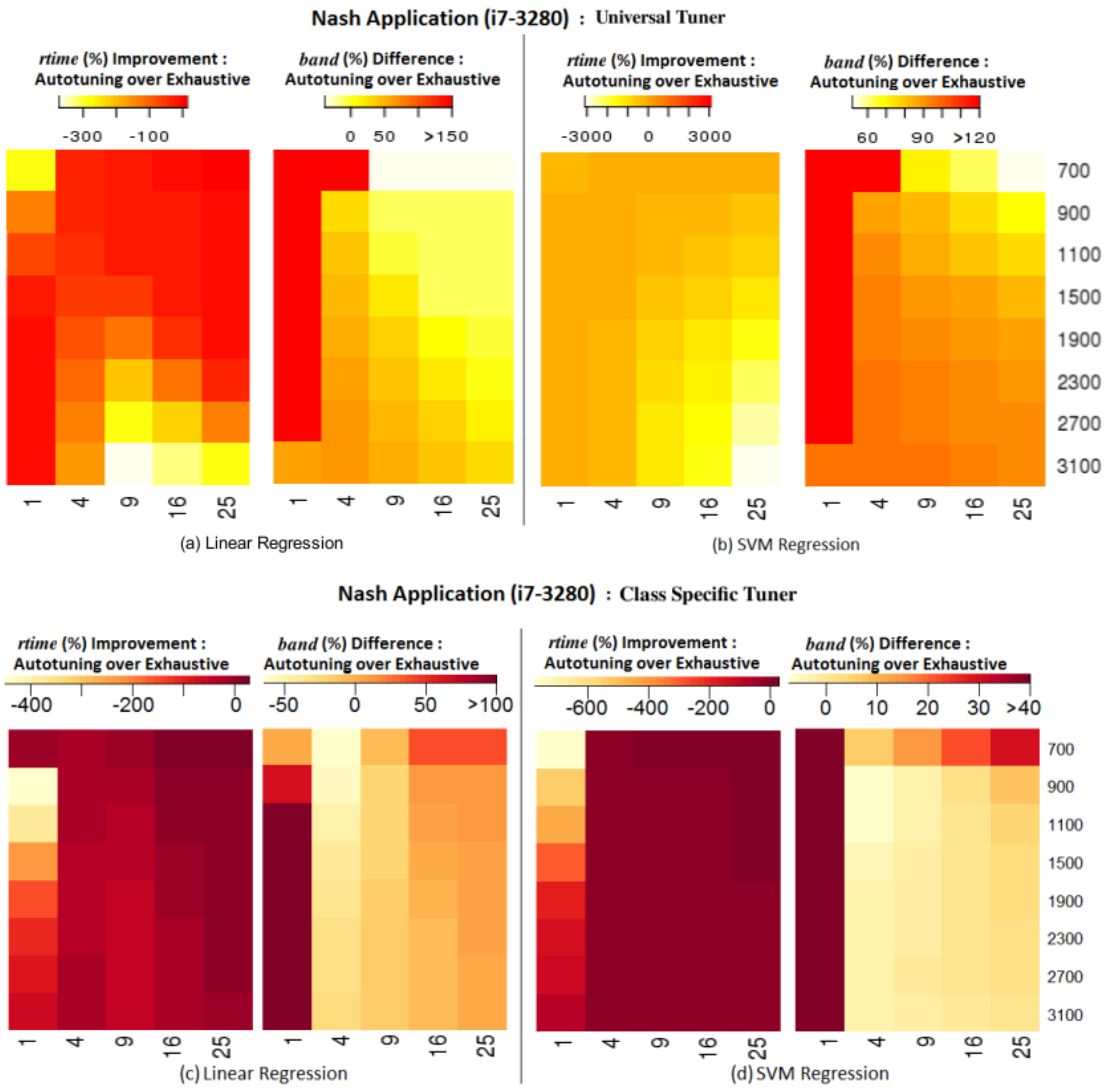


Figure 4.8: Autotuner performance for the Nash application on i7-3280 for class-specific tuners. Each pair of maps shows autotuned performance against the best performance from exhaustive search, and the difference in *band* values used at these points. In all maps the x-axis is *tsize* and the y-axis is *dim*.

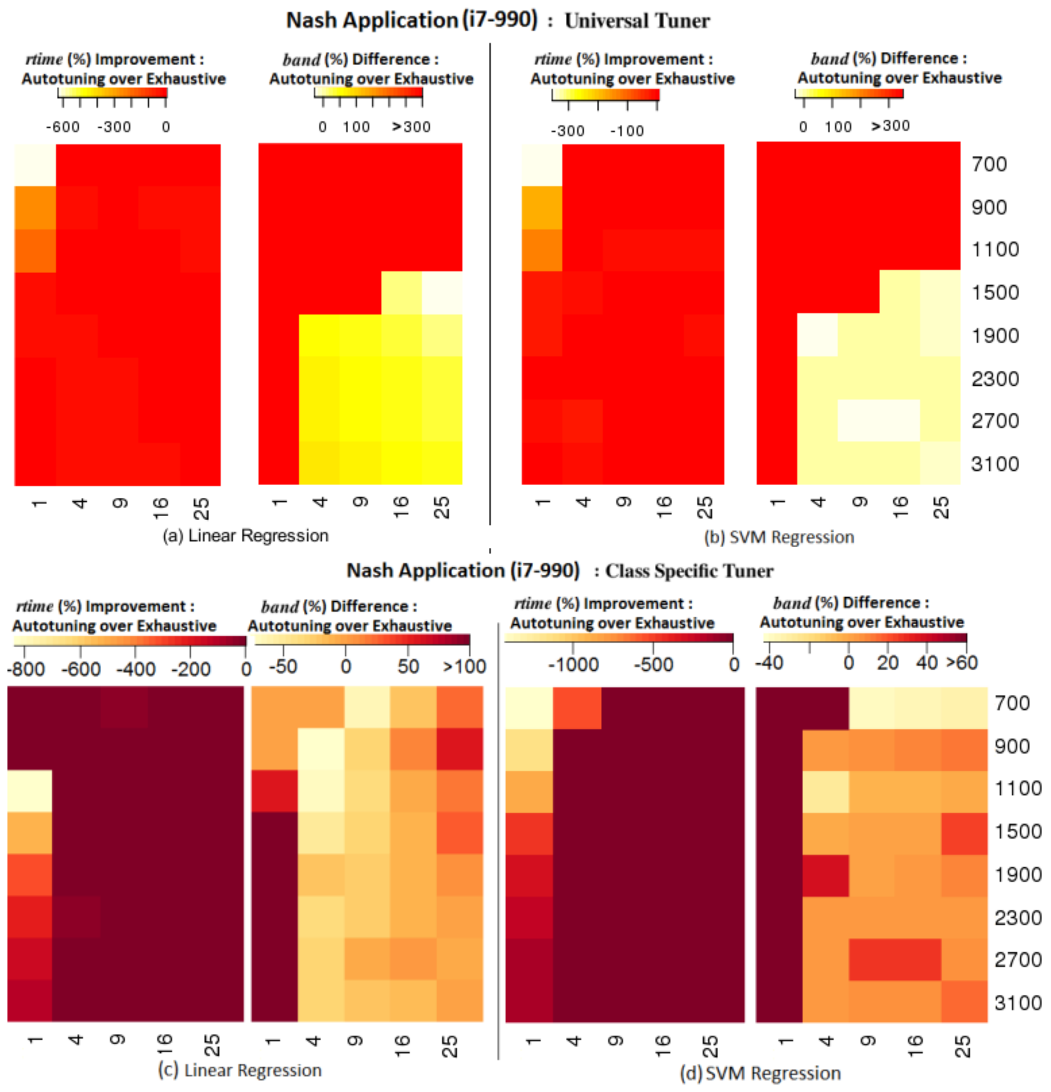


Figure 4.9: Autotuner performance for the Nash application on i7-990 for class-specific tuners. Each pair of maps shows autotuned performance against the best performance from exhaustive search, and the difference in *band* values used at these points. In all maps the x-axis is *tsize* and the y-axis is *dim*.

at higher *dim-tsize*. Linear Regression tuner predicts inaccurate *band* values at lower values of *dim-tsize* compared to Support Vector Machine (SVM) based tuner, which is why it fares better than the SVM model.

The poor performance of both universal tuners in the i7 systems led us to experiment with class specific tuners, as shown in figures 4.8(c) and 4.8(d). There was a high penalty for not being able to predict *band*=-1, leading to an average *rtime* increase by 40% for Linear Regression and by 50% for the SVM model. The class specific tuner also fared better than the universal tuner for the i7-990 and results are shown in figures 4.9(c) and 4.9(d). The Linear Regression model correctly predicted *band*=-1 for *tsize*=1, *dim*=[700, 900]. Overall average *rtime* increased by 59% using Linear model vs 136% for SVM model.

4.3.4 Autotuning Results for Synthetic Application

The autotuning results for the synthetic application across all systems are presented in Figure 4.10, Figure 4.11 and Figure 4.12. For the universal tuner on the i3-540 system, Linear Regression in 4.10(a) performs better than SVM in 4.10(b) as the former manages to predict *band*=-1 where needed and it also has fewer number of points with high difference between predicted and best *band* values. Conversely, in the case of class specific tuning, the SVM tuner of 4.10(d) performs slightly better than the Linear Regression tuner of 4.10(c).

In the i7 systems, there are no Support Vector learner results for class specific tuning. This is because the training data in the Synthetic Application for i7 systems has constant *band* value of -1.

For the i7-990 system, the class specific Linear Regressor correctly predicts *band*=-1, so the difference in *band* values is 0 across all configurations in 4.11(d). Any variation in *rtime* improvement in 4.11(c) is due to *cpu-tile* size.

As before, the Universal tuner fares poorly with the Linear Regression predictor, performing up to 300% worse than the best exhaustive runtime (*ber*) in 4.11(a) and for SVM it goes further down to being 2000% worse in 4.11(b). This is because the best *band* value is -1 in all cases and the universal tuners predict some *band*>-1 which affects the *rtime*. Linear Regression fares better as its predicted *band* values are -1 until *dim*=1500 while for SVM only *dim*=700, *tsize*=1 had a predicted *band* value of -1. Similar results were also observed across the i7-3280 system with the universal tuner faring poorly in figures 4.12(a) and 4.12(b), and the class specific Linear Regression tuner correctly predicting *band*=-1 in 4.12(d). Thus class specific tuning of i7 system performs well for both i7-3280 and i7-990, with an average *rtime* improvement of 1.38% and 5.5% respectively over *ber*.

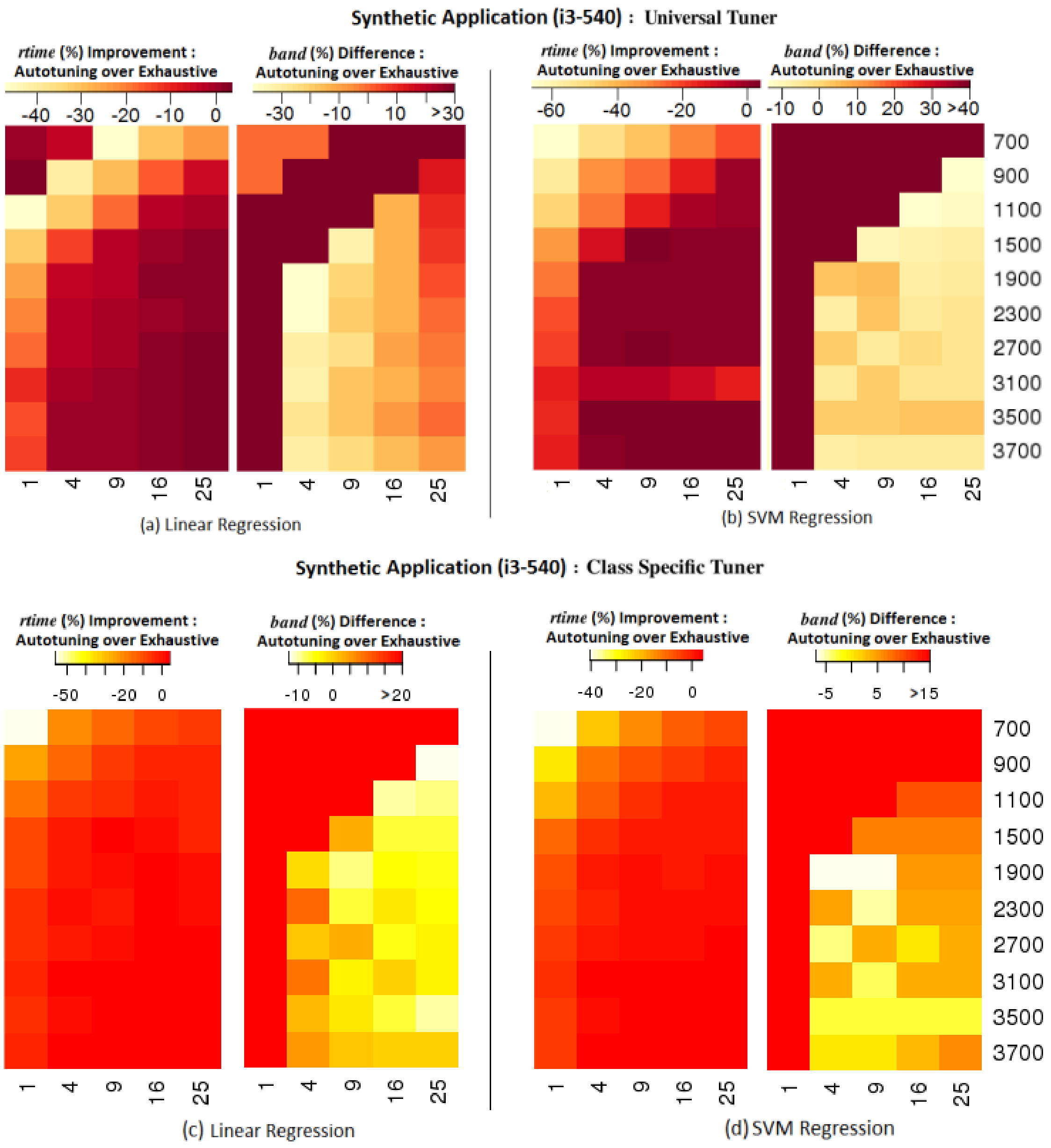


Figure 4.10: Autotuner performance for the synthetic application on the i3-540, using the universal tuner and the class specific tuner. Each pair of maps shows autotuned performance against the best performance from exhaustive search, and the difference in *band* values used at these points. In all of our maps the x-axis is *tsize* and the y-axis is *dim*.

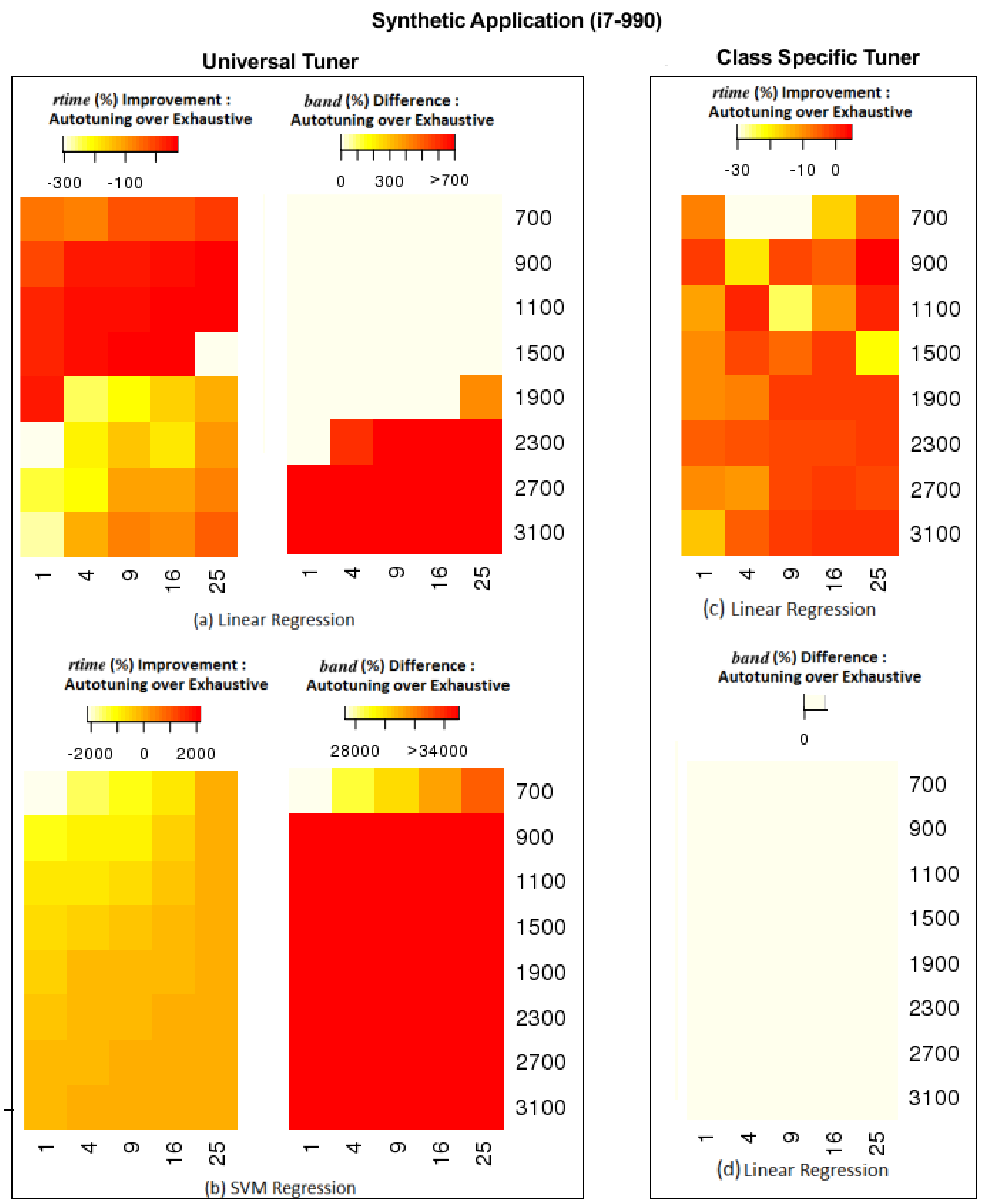


Figure 4.11: Autotuner performance for the synthetic application on the i7-990, using the universal tuner and the class specific tuner. For the universal tuner, each pair of maps shows autotuned performance against the best performance from exhaustive search, and the difference in *band* values used at these points for Linear and Support Vector Predictors. For class specific tuner, only Linear Regressor is used as discussed in 4.3.4 In all of our maps the x-axis is *tsize* and the y-axis is *dim*.

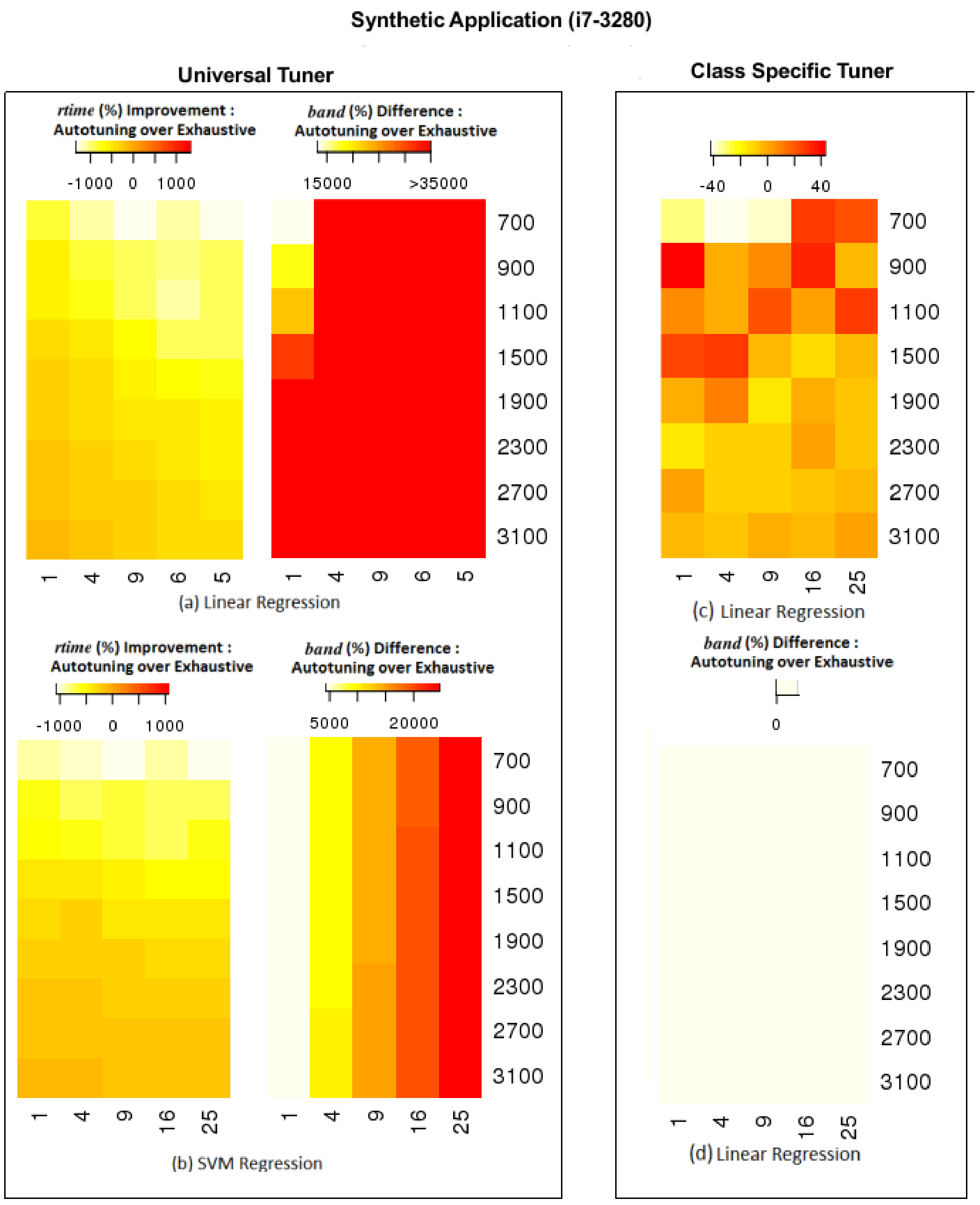


Figure 4.12: Autotuner performance for the synthetic application on the i7-3280, using the universal tuner and the class specific tuner. For the universal tuner, each pair of maps shows autotuned performance against the best performance from exhaustive search, and the difference in *band* values used at these points for Linear and Support Vector Predictors. For class specific tuner, only Linear Regressor is used as discussed in 4.3.4. In all of our maps the x-axis is *tsize* and the y-axis is *dim*.

Table 4.2: Optimal speed-up across applications and systems

System	Optimal Speedup	Tuner	Model	% of Optimal
i3-540	6.34x	Universal	SVM	87.18%
			LR	85.80%
		Class Specific	SVM	95.4%
			LR	94.5%
i7-990	7.13x	Universal	SVM	40.18%
			LR	57.61%
		Class Specific	SVM	N/A
			LR	89.5%
i7-3280	37x	Universal	SVM	39.07%
			LR	48.23%
		Class Specific	SVM	N/A
			LR	91.7%

4.3.5 Autotuning Results for Sequence Comparison Application

The autotuning perfomance of the Sequence Comparison application is presented in Figure 4.13. The universal tuner fared poorly in all cases and the SVM Regression tuner was worse than the Linear Regression tuner, with the former on average performing 24 times worse than the best points and the latter being 12 times worse on average.

None of the systems had class specific SVM predictor because the training data in the Sequence Comparison application had constant *band* value of -1. The class specific Linear Regression tuners predicted *band*=-1, which was correct across all systems. Thus *rtime* depended on accurately predicted *tile* values. For the i7-3280, *rtime* averaged only 4.7% more than *ber* and in both i7-990 and i3-540, *rtime* averaged 7% more than *ber*.

4.3.6 Summary

Table 4.2 shows, for each system and autotuning strategy, the average speed-up across all applications found by exhaustive search, and the percentage of this achieved by the autotuner. SVM class-specific tuners for i7 systems were not relevant for non Nash applications as the *band* values were constant at -1 in all case. This trivial case causes SVM to fail but would be easy to screen out in a production system.

4.4 Conclusion

In this chapter we discussed our autotuning strategy for 2D wavefront applications and the performance of our autotuner. We first explored the runtime performance of each application across three different systems for various combinations of input parameter values. Then we

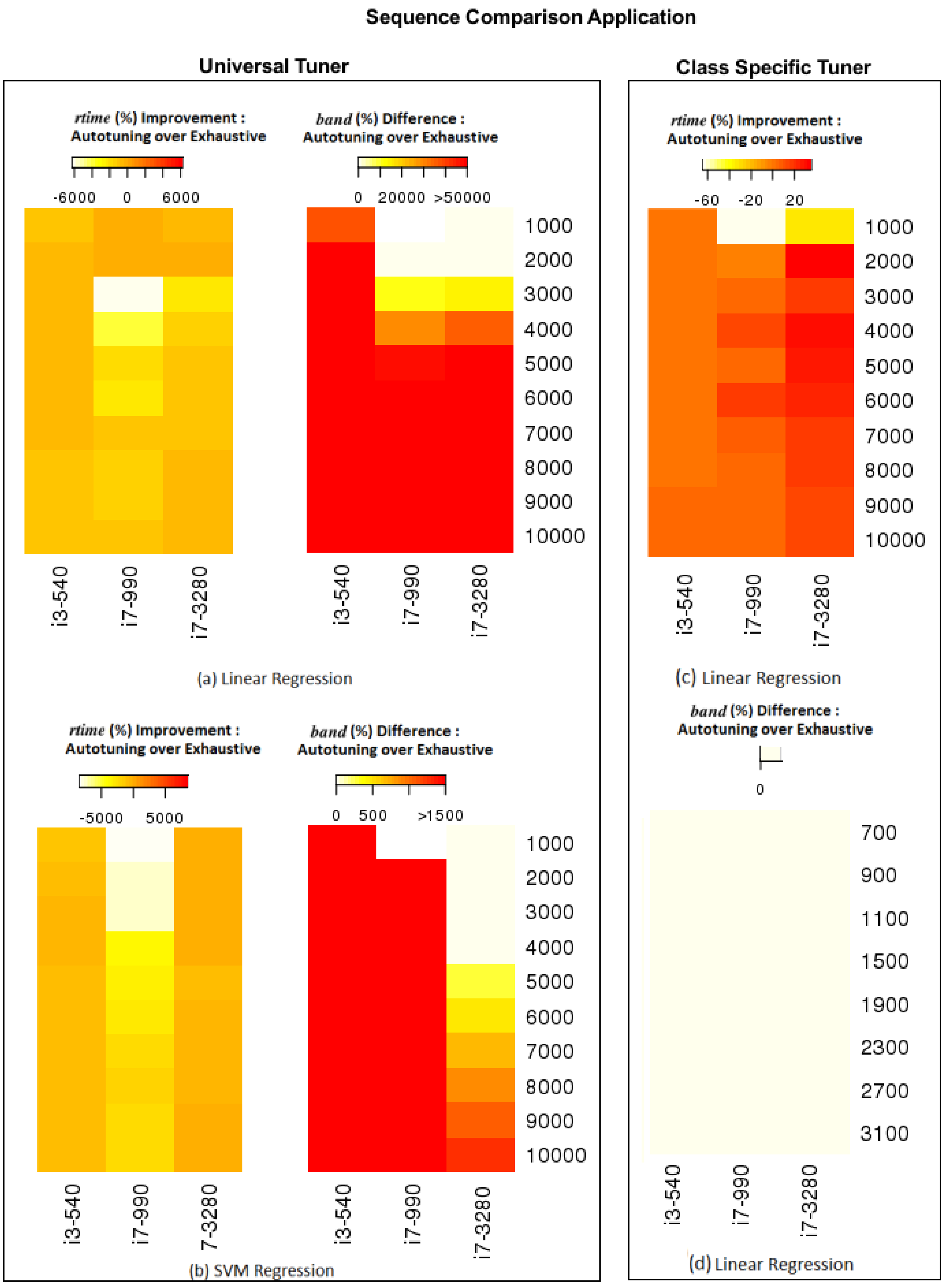


Figure 4.13: Autotuner performance for the sequence comparison application for all systems, using the universal tuner and the class specific tuner. For the universal tuner, each pair of maps shows autotuned performance against the best performance from exhaustive search, and the difference in *band* values used at these points for Linear and Support Vector Predictors. For class specific tuner, only Linear Regressor is used as before. In all of our maps the x-axis represents the systems and the y-axis is *dim*. It should be noted that the sequence comparison application has constant *tsize*, hence there is no variation for this parameter.

selected a subset of the exhaustive search results to train our learning models. Evaluation of our tuner was then carried out on a separate test set.

Our first tuning decision was a simple sequential/parallel choice. Where a parallel solution was selected, we observed that well chosen settings for the *tile* and *band* parameters produced very significant improvements in performance, and correspondingly that poorly chosen settings resulted in performance which was sub-optimal. Our autotuning experiments showed SVM regression outperformed Linear Regression on the i3-540 system, but the reverse was true for the i7 systems.

Class-specific tuners equaled or outperformed the universal tuner in all situations. This raised a new challenge. Our classes here were defined intuitively, but a fully automated class-specific solution would have to perform this classification directly. For that to occur, firstly the synthetic training set has to be large enough to encompass a wide range of input parameter values of task granularity and problem size, and the learning model has to perform better than the ones we had employed. These issues are addressed in the next chapter.

Chapter 5

Multiple GPU Autotuning : 2D Wavefronts

5.1 Introduction

We now consider the additional challenge of autotuning wavefronts for systems with two GPUs. Once again, our goals are to understand the relationship between settings of the internally tunable implementation parameters and performance, and to use machine learning techniques to control the automatic setting of these parameters. We first explain our experimental program in section 5.2. The first phase of our experimental program deals with training our model, using the synthetic wavefront application. The second phase applies the learned model to real, previously unseen wavefront applications. We then analyze the results of our exhaustive search in section 5.3 and examine the details of our autotuning in subsection 5.3.2. Finally we discuss the evaluation of our learned models in subsection 5.3.3.

5.2 Experimental Program

In this section we discuss the training phase of our autotuner followed by the evaluation phase and the systems on which our experiments were carried out.

5.2.1 Training Phase

Training is conducted with instances of a synthetically generated wavefront application, instead of using training instances from all wavefront applications as was done for the *universal* tuner in chapter 4. This is parameterizable across a wide range of size and granularities. While the synthetic application of the previous chapter simulated real world applications whose task granularity lied in mid-range, the current synthetic application simulates the entire range of fine grain to coarse grain applications. This allowed us to exclude training

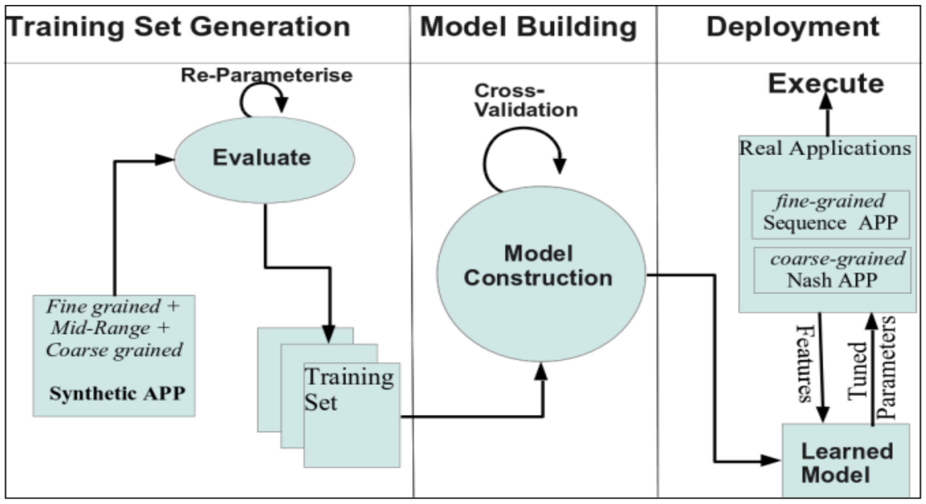


Figure 5.1: Machine Learning Strategy : The training set is created by selecting high performing instances from an exhaustive parameterized search of the synthetic wavefront application. Decision tree models are built from the training set and cross validated. In deployment, the model is passed features of the previously unseen application and returns appropriate tuning parameter settings.

instances from real world applications, in contrast to the previous universal tuners that were trained on instances from two real world applications.

5.2.1.1 Parameter Space

Table 5.1: Parameter Ranges

Parameter	**Range**
dim	500 to 3100
tsize	10 to 12000
dsize	1, 3, 5
cpu-tile	1, 2, 4, 8, 10
band	-1 to 2**dim*-1
gpu-count	0, 1, 2
halo	-1 to 0.5*(length of first offloaded diagonal)
gpu-tile	1, 4, 8, 11, 16, 21, 25

In order to gain insights into the shape of the performance space and trade-offs, we first conduct an exhaustive evaluation of our synthetic application, across a range of settings for the input and output parameters, as listed in table 5.1. The values selected for the input parameter space (*dim*, *tsize* and *dsize*) are meant to be representative of real world applications. For some tunable parameters, our experiments were carried out within the range of values that are restricted by hardware and framework implementation. These are *gpu-count* and *gpu-tile*. Other tunable parameters like *band* and *halo* are dependent on values of the input parameter space. The *cpu-tile* parameter range was chosen empirically after observing tiling had no noticeable effect on performance for *cpu-tile* ≥ 10 and $500 \leq$ *dim* ≤ 3100.

dim is straightforward. *tsize* is measured in units of the execution time of a single iteration of the synthetic kernel function on a single CPU core. For *dim* values ranging 500 to 3100, a single iteration of the coarse grained Nash Equilibrium application takes the same time as 750 iterations of our synthetic application (*tsize*=750) while the fine grained Biological Sequence Comparison application takes half the time of a single iteration of the synthetic application (*tsize*=0.5).

The data structure for each element in our synthetic application consists of two int variables and a varying number of floats, controlled by *dsize*. For example, *dsize=5* means size of each element is 8+5∗8=48 bytes and so on. Nash Equilibrium application with its two int and four float variables has a size of 40 bytes, so that *dsize*=4. Biological Sequence Comparison application with its 1 int and 2 char variables has a size of 6 bytes but *dsize*=0 as it has no floating point data.

Values of parameters like *dim, tsize, band, halo* are spaced irregularly to avoid any cyclic pattern and incorporate a degree of randomness. The best performing values are used in training our learning models.

To simplify modeling, we have overloaded the *band* and *halo* parameters to encode *gpu-count*. As before, a *band* of n means that $2n+1$ diagonals in total are assigned to the GPU, a *band* of -1 means that the GPU is not to be used. Larger band values mean that at least one GPU is used, with a non-negative *halo* size indicating the *gpu-count* is 2.

To enable us to explore the parameter space within a reasonable time, we set a threshold limit of 90 seconds on the runtime (*rtime*) for any execution. This has no impact on our tuning since any point that exceeds this threshold limit is already a very bad configuration which would not be selected as a training example. We removed the threshold in collecting points for our serial baseline in order to correctly compute performance improvement.

5.2.1.2 Autotuning Strategies

We use decision trees to derive our learning model, using training data drawn from the synthetic application. Training sets are created by subsetting the exhaustive search data as follows: firstly a subset of the problem instances (i.e., by *dim*, *tsize* and *dsize*) are selected by regular sampling; then the best five performance points for these instances (by tunable parameter values) are added to the training set. The intuition is that these should be representative of the good decisions we wish to embed in our models. Initial evaluation is done through *cross-validation*, meaning evaluation is conducted on instances of the synthetic application which are omitted from the training set at the first step, to avoid over-fitting. We explore different configurations of the learning model to obtain test results that are at least 90% accurate. This model is then applied to the real applications. As before, this procedure is repeated independently for each system, in line with a scenario which would see the software trained "in the factory".

During training, we first build a binary SVM based predictor to decide whether or not to exploit parallelism. For those cases in which parallelism is predicted to be beneficial we then apply and evaluate two machine learning heuristics, based on M5P Decision Tree and REP Tree [37]. Previous work [76] found simple Linear Regression models lacking, and upon exploring different learning models and analyzing the exhaustive search space, we found the decision trees to be most accurate in predicting optimal values for our tunable parameters. This is explained later in subsection 5.3.2.

5.2.2 Evaluation Phase

We evaluated the performance of our learned model on two real world wavefront applications, discussed earlier in section 2.6. They are the coarse grained Nash application and the fine grained Sequence Comparison application.

The input parameter values of these real world applications map to our synthetic scale as follows: one iteration of the Nash Equilibrium application corresponds to a *tsize*=750 with data granularity of *dsize*=4. One iteration of the Biological Sequence Comparison application has *tsize*=0.5 and *dsize*=0 since there is no floating point data.

5.2.3 Platforms

Our three experimental systems are described in table 5.2. 'HT' stands for hyper-threaded CPU cores and 'CU' refers to the GPU compute units. The single GPU i3-540 system is the same as in chapter 4. The two i7 systems are however different as these are multiple GPU systems compared to the single GPU i7 systems of the previous chapter. As earlier, our choice of these contrasting systems is meant to be representative of the variety found in real world heterogeneous systems. The i3 system is the slowest among the three systems with 1.2 GHz clock speed and a relatively fast GTX 480 GPU. The i7-2600K system has faster CPU cores at 1.6 GHz and four fast GTX 590 GPUs (only two are used in our work). The i7-3820 has the fastest CPU cores at 3.6 GHz and while the two Tesla GPU cores are slightly slower at 1147 MHz compared to GTX 590, their memory capacity is the highest among our GPUs.

We measure runtime of the whole program execution using wall clock timers in the host program, averaging across three runs (which exhibited low variance of less than .01).

Table 5.2: Experimental Systems

System	Freq (Mhz)	Cores (HT)	Mem (GB)	GPU	Freq (Mhz)	CU	Mem (GB)
i3-540	1200	4	4	GTX 480	1401	15	1.6
i7-2600K	1600	8	8	4 x (GTX 590)	1215	16	1.6
i7-3820	3601	8	16	Tesla C2070, Tesla C2075	1147	14	6.4

5.3 Results and Analysis

In subsection 5.3.1 we investigate the characteristics of the search space created by our synthetic training application, and explore the resulting model. In subsection 5.3.2 we evaluate the model on real world applications.

5.3.1 Training : Exhaustive Search Results

We now present the results of our exhaustive search space exploration of the synthetic application across all three systems. This subsection contains an analysis of the best performing points found from exhaustive search. We explore the GPU usage (*band* values) at these points, and how this is affected by problem features (size, task and data granularity) and system characteristics. For optimal points that had multiple-GPU usage, we analyze the extent of *halo* region between dual GPUs. Finally we analyze the effects of *gpu-tiling* at these points.

Next we analyze the performance of these optimal points against simple schemes such as computing everything serially, computing in parallel with no GPU usage and computing every element inside a GPU. This is to compare how our framework, with its sophisticated mechanism of partitioning computation (*band* and *halo*) and its built-in optimization techniques (*cpu-tile* and *gpu-tile*) fares against such simple schemes. Then we compare the *rtime* averaged across all possible tunable parameter configurations against the *rtime* of our optimal points. This is to provide an estimate of how well a randomly chosen configuration would perform against our exhaustively searched optimal configuration. If the averaged *rtime* and best *rtime* are within few percentage difference, then we might as well randomly pick some configuration to obtain good performance.

This leads us to the concept of sensitivity, that forms the last portion of this subsection. While the averages provide an estimate, we explore in detail the performance of every configuration against the optimal configuration in our search space and observe if there are many or few configurations with *rtime* close to the best *rtime*, forming a thick or thin cluster around the best configuration. Having many points around the best performing one means we can randomly choose some point which will perform well and our search space is thus insensitive to changes in tunable parameter values. The reverse is true for a sensitive search space. We measure sensitivity by analyzing the *rtime* from all tunable parameter configurations across every combination of input parameter (*dim* - *tsize* - *dsize*) values in the form of violin plots. This is explained in detail in subsubsection 5.3.1.4.

5.3.1.1 Optimal Performance Points

Figure 5.2 presents a set of four heatmaps for the two i7 systems with multiple GPUs and two heatmaps for the i3 system with a single GPU. In all maps *tsize* and *dim* are the x and y axes. The heat maps illustrate the values of *band* and *halo* (for multi GPU systems) that result in the fastest execution time. The upper half heat maps correspond to *dsize*=1 (element size=16 bytes) and lower half with *dsize*=5 (element size=48 bytes). We analyze the results based on the trade-off considerations mentioned earlier in section 3.4.

GPU Usage (*band*) :

From the maps it is clear that computing on the GPU ($band \geq 0$) becomes favorable when task granularity exceeds a certain threshold and this threshold varies depending on the problem size, data size and the hardware. We consider each of these effects below.

- **Problem Size and Task Granularity Effect** : GPU use becomes feasible when there is sufficient parallelism available in the form of concurrently computable elements in each successive wavefront diagonal and the computation is coarse enough to overcome the communication overhead between CPU and GPU. Across all heatmaps, GPU use can be seen for problem sizes (*dim*) $\geq$ 1900 and task sizes (*tsize*) $\geq$ 2000. In general, the *band* value increases with increase in *dim* and *tsize*, though there is an exception to this rule when multiple GPUs ($halo \geq 0$) are used. We discuss this later in this subsection.

- **Data Size Effect** : The effect of *dsize* can be seen in all three systems, where the 48 bytes sized elements make GPU use costly due to data transfer overhead to/from GPU as previously discussed in 3.4. When *dsize*=5, GPU usage is feasible at high values of problem size and task granularity with ($tsize \geq 2000$, $dim \geq 1900$) in the i7 systems and ($tsize \geq 700$, $dim \geq 1100$) in the i3 system. This threshold is low for *dsize*=1 with GPU use becoming feasible at ($tsize \geq 1000$, $dim \geq 1500$) in the i7 systems and ($tsize \geq 250$, $dim \geq 1100$) in the i3 system.

- **Hardware Effect** : Consider the case of *dsize*=1 (element size=16 bytes) for the i7 systems with fast CPU cores, where the GPU is used from $tsize \geq 500$ and $dim \geq 1900$ onwards. This differs from the i3 system with its slower CPU cores where GPU is used

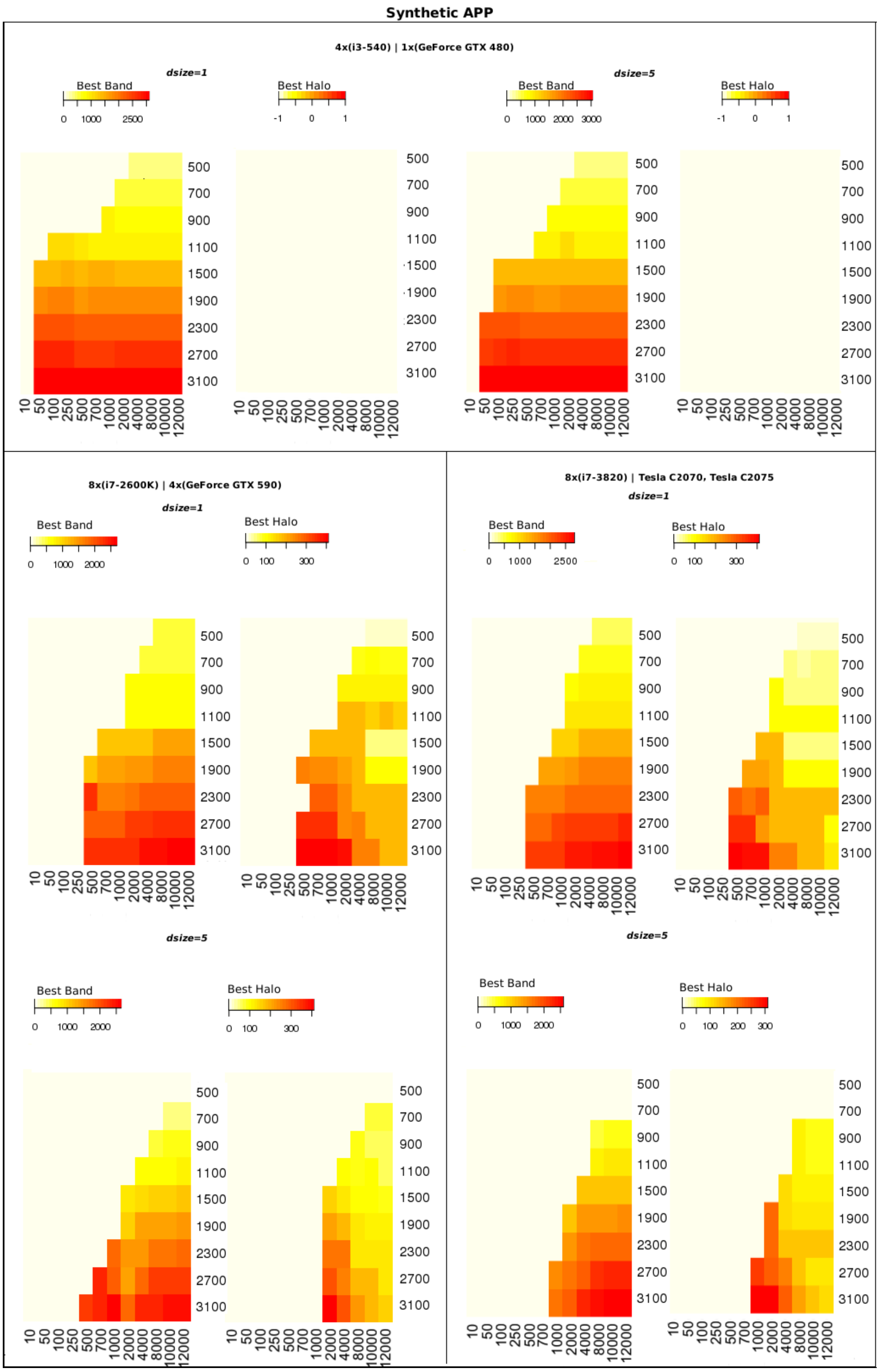


Figure 5.2: Heatmaps illustrate the *band* and *halo* values at the best performing points from our exhaustive search across three systems for an element size of 16 bytes (*dsize*=1; 1 float and 2 ints) and 48 bytes (*dsize*=5; 5 floats and 2 ints). The i3 system is a single GPU system, hence no halo heat map is shown. In all maps the x-axis is *tsize*, indicating kernel task granularity and the y-axis is *dim*, indicating problem size.

at a lower threshold of *tsize* $\geq$ 100 and *dim*$\geq$ 1100. A slower CPU coupled to a faster GPU means most of the data should be offloaded onto the GPU and this is validated by the above result.

Multi-GPU Usage (*halo*) :

As discussed earlier in section 3.4, dual GPUs have the additional communication overhead of *halo* swapping that gets more expensive for large data sizes and communication cost is higher for fine grained tasks. However if the task granularity is above a certain threshold, the redundant computations in the *halo* regions should dominate over the communication cost. Problem size also plays a part in multi-GPU usage as larger problems have more concurrently computable elements that can be split among multiple GPUs.

- **Data Size and Task Granularity Effect** : In both multi-GPU i7 systems, we see *halo* $>$ -1 for *tsize* $\geq$ 500 when *dsize* $=$ 1. This means for small data granularity and sufficiently large computation granularity, there is little communication overhead in swapping boundary elements between two GPUs making dual GPU use effective. However, this threshold increases to *tsize* $\geq$ 2000 when *dsize* $=$ 5. This confirms our hypothesis that increase in communication overhead is due to the increase in data size, rendering multi-GPU usage unfeasible.

  We also note that the best *halo* values start decreasing steadily from a peak value. For example in case of the i7 systems at *dim*=[2700, 3100] and *dsize*=1, *halo* values peak in the regions of 500 $\geq$ *tsize* $\geq$ 1000 and then gradually decreases. This validates our reasoning that *halo* sizes for the multi-GPU systems are higher when *tsize* values exceed a certain threshold owing to the trade-off between redundant computation cost and lesser communication cost.

- **Problem Size Effect** : The problem size naturally determines the extent of *halo* regions and the heatmaps show a steady increase in *halo* values for corresponding increase in *dim* values until a saturation point that depends on *dsize* and *tsize*, as discussed above.

- **Hardware Effect** : *halo* is also affected by hardware as seen in the heatmaps for the i7 system (and is non-existent for single GPU i3 system). The i7-3820 CPU has higher operating frequency than the i7-2600K CPU (3.9 vs 3.8 Ghz) and greater L2+L3 cache (11 MB vs 9 MB) meaning i7-3820 should have more CPU computation relative to the i7 2600K. Besides the GTX cards have faster and higher number of compute units (see table 5.2). This is seen from the lower *halo* values for the 3820 system at *tsizes* 2000 to 12000 but there is an exception when *dsize*=5 and *tsize*=1000 where the Tesla cards of the 3820 system take up more computation relative to the GTX cards of the 2600K system.

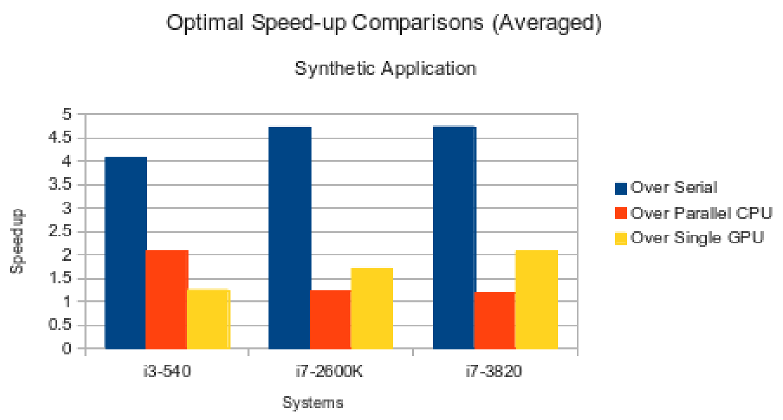


Figure 5.3: Bars illustrate the speedup of the heat-map points from figure 5.2 over serial, parallel CPU and single GPU baselines.

Tiling :

We conclude the heatmap observations by noting that GPU tiling was not beneficial in our search space. This was because tiled GPU performed better than the untiled GPU implementation in cases where the communication costs dominated over computation costs, $tsize < 50$. However in these situations, the CPU only parallel implementation dominated over any GPU based implementation due to the additional overhead incurred from starting the GPU.

5.3.1.2 Comparison with Simple Schemes

Next we investigate the quality of these heatmap points. We compute the average of the best *rtime* across all input parameter values (*dim*-*tsize*-*dsize*) from our exhaustive search and compare it against the best performing points from three simple schemes of carrying out computation a) serially in the CPU, b) in parallel across all CPU cores with no GPU phase and c) entirely in the GPU (figure 5.3)

The quality of our heat map points or optimal points from exhaustive search is superior to the best performing points from simple schemes. This is observed from the 4 - 4.5 times speedup over serial scheme, 1.05 - 2 times speedup over parallel CPU scheme and 1 - 2 times speedup over single GPU scheme.

We note that in case of the i7 systems, on average, doing everything on the GPU, is worse than doing everything on the CPU. This is because the fast CPU outperforms the GPU by a large margin for low task granularity points (up to 10 times for $tsize \leq 100$, $dim \leq 1100$).

5.3.1.3 Average Case Comparison

The next comparison evaluates optimal heatmap points against average behavior. This is seen in detail in figure 5.4, which representing the best exhaustive runtime (abbreviated to

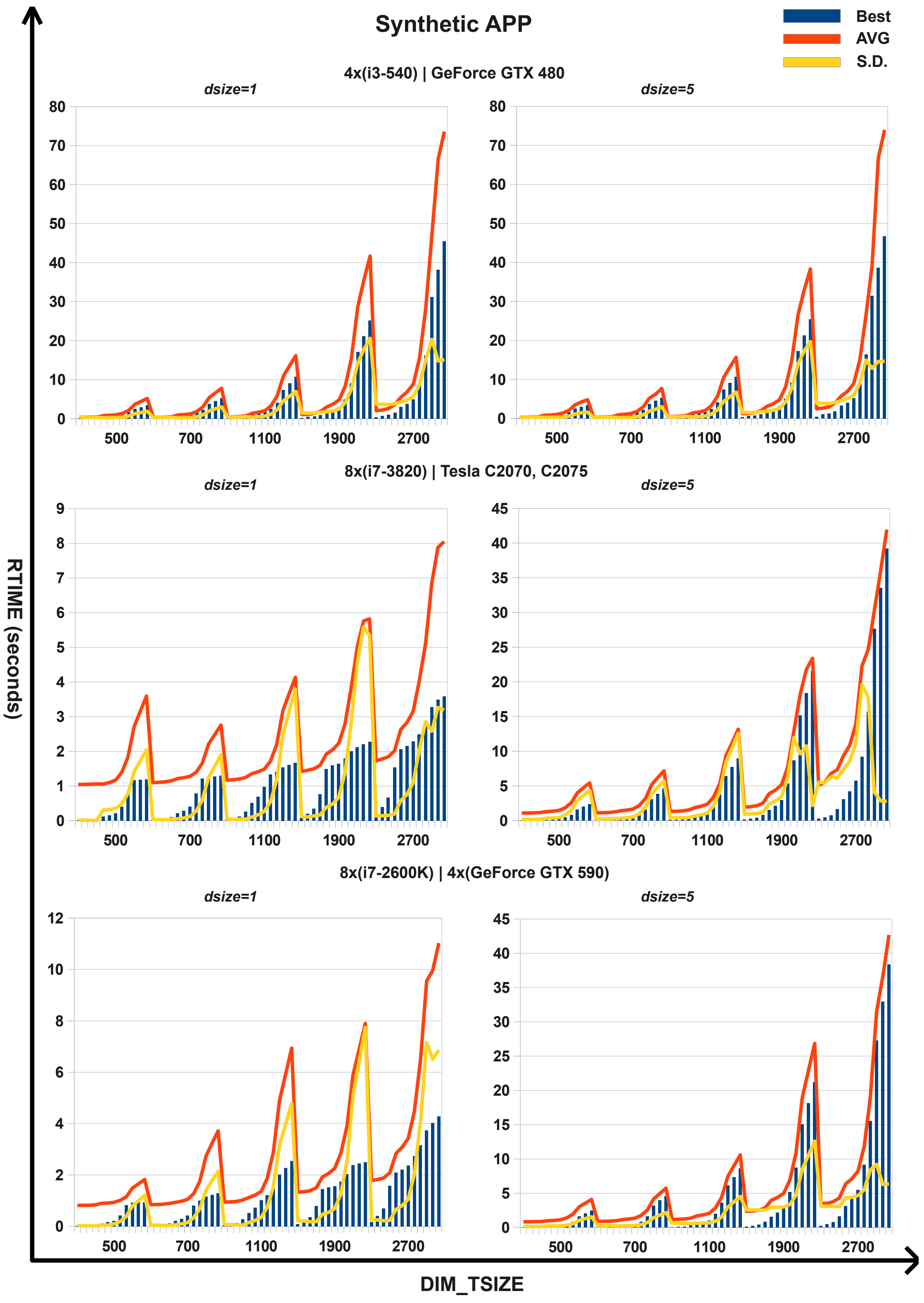


Figure 5.4: Average case comparison for the Synthetic Application. The x-axis is *dim-tsize*, indicating groups of problem sizes whose kernel task granularity varies from 10 to 12K and the y-axis is *rtime*, indicating actual runtime. Best is the best exhaustive *rtime* (*ber*), AVG is the average *rtime* from all configurations, S.D. is the standard deviation from average. *dsize* refers to the number of floats in our synthetic data structure containing 2 int variables. Total element size = 16 bytes (*dsize*=1; 1 float and 2 ints) and 48 bytes (*dsize*=5; 5 floats and 2 ints)

ber) and the runtime (*rtime*) averaged across all possible combinations of tunable parameters. The figure includes corresponding standard deviations. The x-axis shows groups of *dim-tsize* with *dim* varying 500 to 2700 with each *dim* grouping *tsize* varying from 10 to 12000. The y-axis is the *rtime* in seconds. Both halves show the performance across all three systems when element size=16 bytes and 48 bytes respectively. For *dsize*=1 (element size=16 bytes), the *ber* is 1.5-2 times faster than the average. The standard deviation steadily increases from *dim*=500 to *dim*=1900 due to the widening gap between the best performing and worst performing points. At *dim*=2700 there is a sharp drop as the *rtime* values exceeded our 90 second threshold. These points were excluded from the average. In case of *dsize*=5 (element size=48 bytes), the gap between *ber* and average *rtime* for *dim*=2700 at *tsize*=[8K, 10K, 12K] narrows down to being just 20%. With higher *dsize*, the GPU overheads become larger. So the best *rtime* points which normally belong to dual GPU + CPU configurations, come closer to the average *rtime*. With higher *dsize* more points also get excluded for exceeding our 90 second threshold so the average doesn't include many poor configurations. This does not affect autotuning since poor points are not used for training.

5.3.1.4 Sensitivity Analysis

We now explore how sensitive the best points are to changes in parameter values. Higher sensitivity would indicate that finding these points is challenging, whereas low sensitivity would indicate that simple random methods might suffice. We present this in the form of violin plots (a combination of box-plots and kernel densities) where the median value is represented as the white dot, with the best and worst points at the extreme ends.

i7-2600K Sensitivity We begin with observing the entire search space distribution for the synthetic application in the i7-2600K system, shown in Figure 5.5, Figure 5.6 and Figure 5.7 which correspond to the three broad category of groupings by data size or *dsize* of {1,3,5}.

Along the x-axis, violin plots are grouped according to a nested scheme of *dim* and *tsize*. Inside each of these groups, the kernel task granularity or *tsize* varies from 100 to 12000 for problem size that ranges from 500 to 3100. The y-axis shows the actual execution time or *rtime*. We now examine the effects of each of these input parameters on the sensitivity of this search space.

- **Data Granularity Effect** : With increase in *dsize* from 1 to either 3 or 5, the sensitivity of points decreases with the best performing points in each grouping of *dim-tsize* falling from being 7-8 times faster than the worst points to being 3-4 times faster than the worst points. This happens due to the increased communication overhead with larger data that to some extent negates the gains of using single/multiple GPUs as discussed earlier in section 3.4. The median value also gets closer to the base.

- **Task Granularity Effect** : Another observation is that low task granularity points

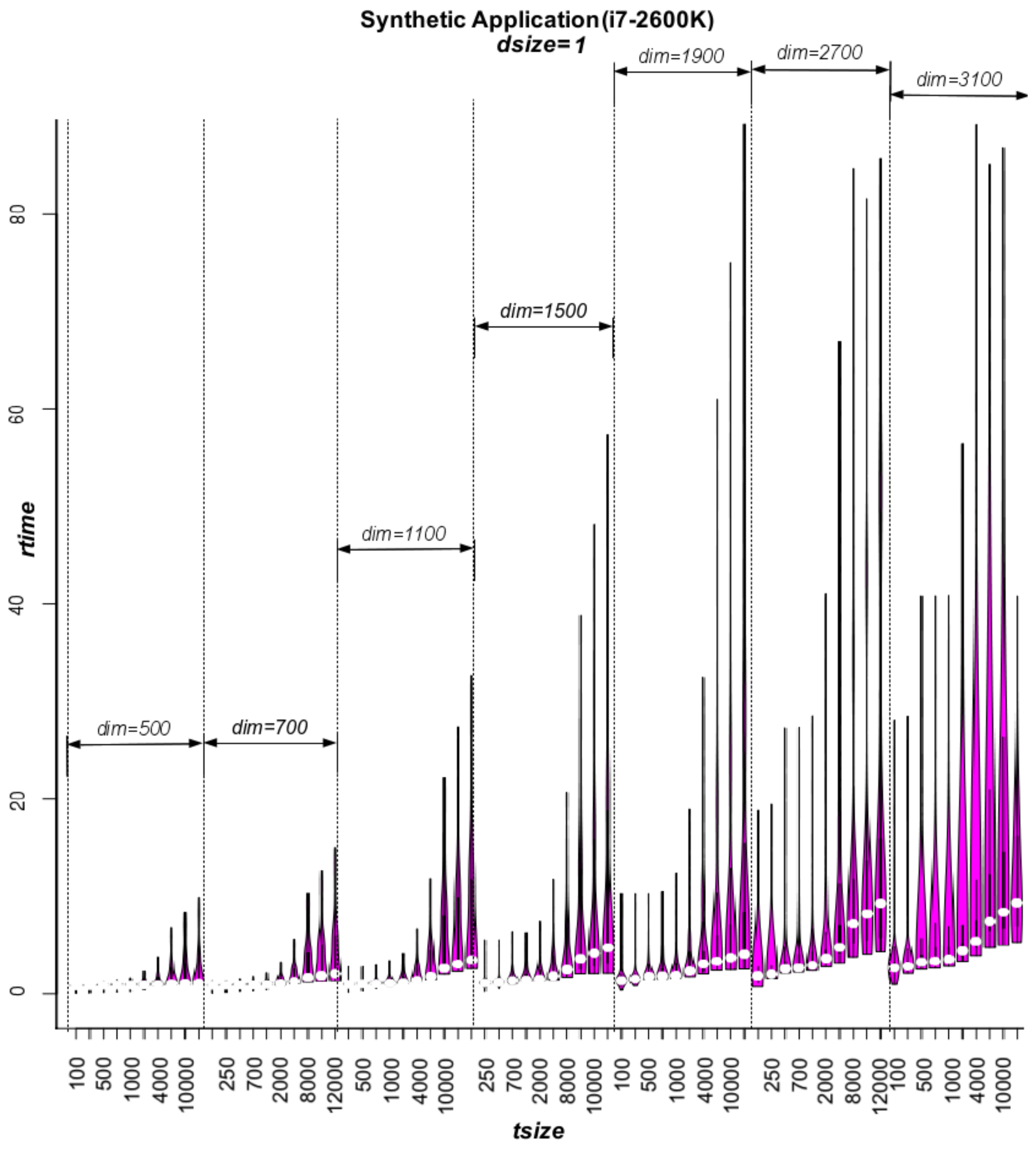


Figure 5.5: Violin plots showing dispersion of all configurations for the synthetic application in the i7-2600K system. The best points are at the base and the white spots are the medians. The x-axis is *tsize*, indicating kernel task granularity. The y-axis is *rtime*, indicating actual execution time. The violin plots cover the entire search space with *dim* varying from 500 to 3100, *tsize* from 10 to 12K and *dsize* = 1. Segments of this space are examined in greater detail in Figure 5.8.

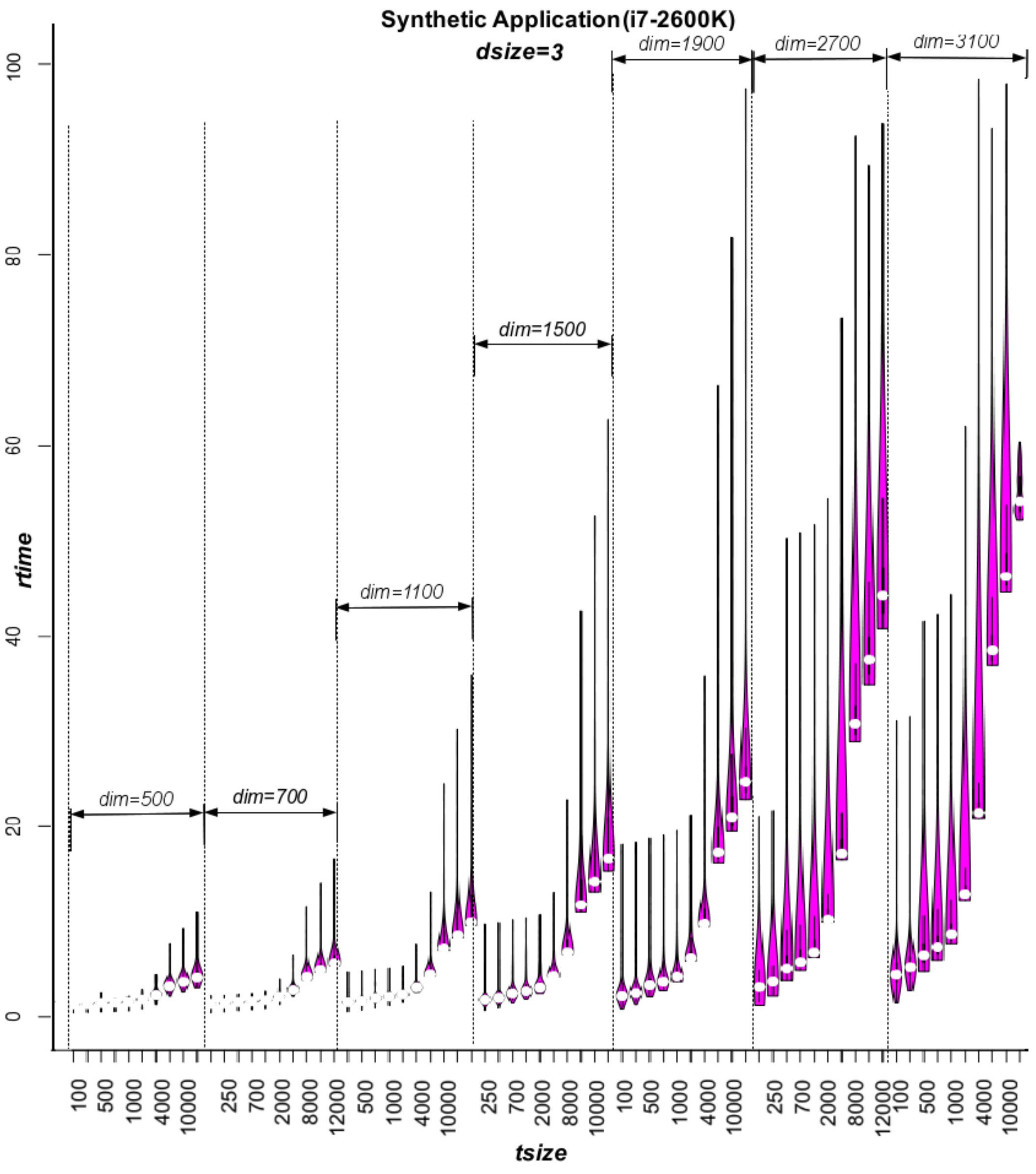


Figure 5.6: Violin plots showing dispersion of all configurations for the synthetic application in the i7-2600K system. The best points are at the base and the white spots are the medians. The x-axis is *tsize*, indicating kernel task granularity. The y-axis is *rtime*, indicating actual execution time. The violin plots cover the entire search space with *dim* varying from 500 to 3100, *tsize* from 10 to 12K and *dsize* = 3. Segments of this space are examined in greater detail in Figure 5.8.

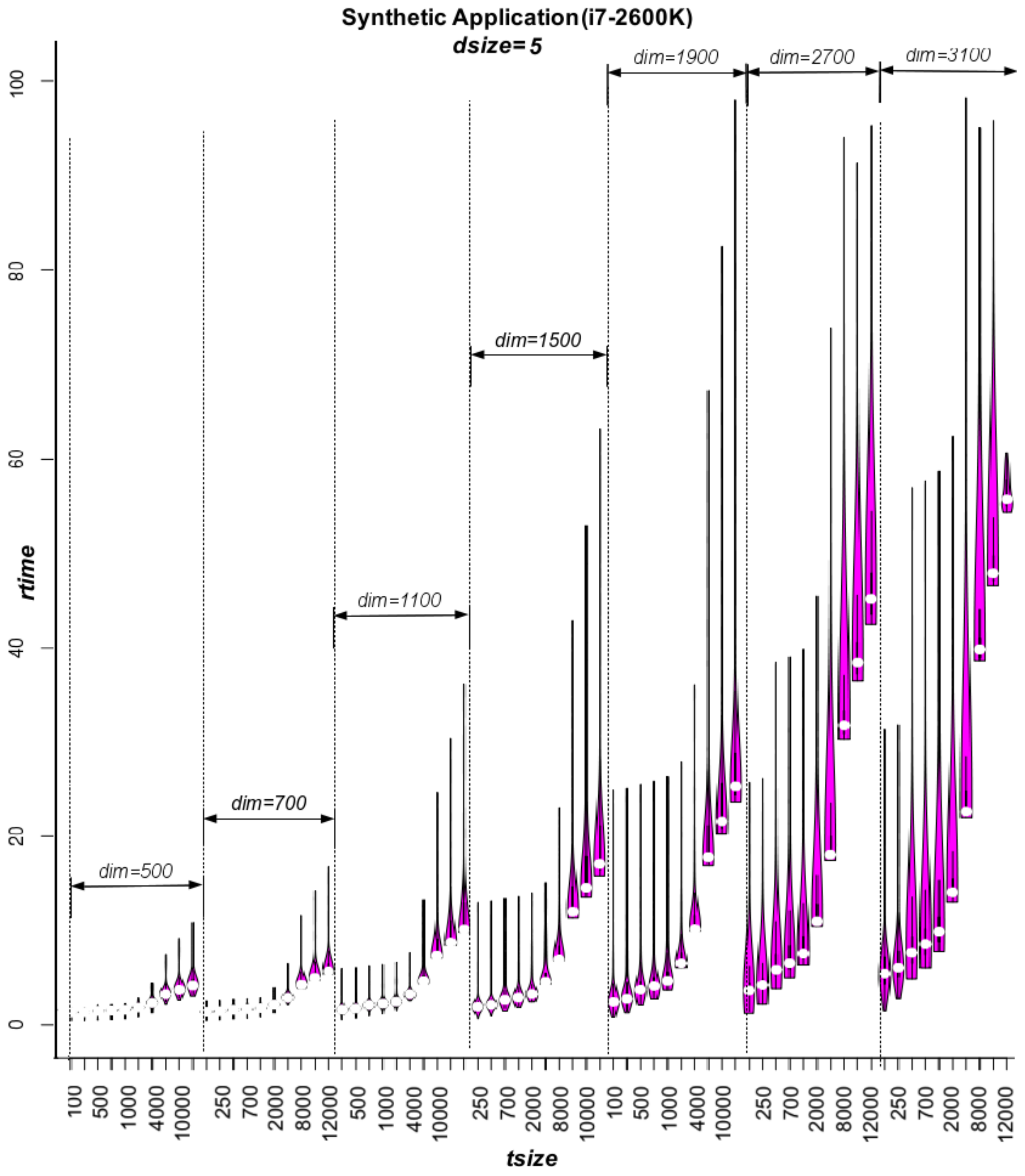


Figure 5.7: Violin plots showing dispersion of all configurations for the synthetic application in the i7-2600K system. The best points are at the base and the white spots are the medians. The x-axis is *tsize*, indicating kernel task granularity. The y-axis is *rtime*, indicating actual execution time. The violin plots cover the entire search space with *dim* varying from 500 to 3100, *tsize* from 10 to 12K and *dsize* = 5. Segments of this space are examined in greater detail in figure 5.8

are more sensitive in general due to communication dominating computation, requiring careful selection of *band* values. However, at the other extreme, sensitivity of very coarse tasks (*tsize* $\geq$8000) gradually increases as seen from the rise in median (the white spots) from the base of the violin plots. Here the best points usually have *halo* ≥ 0 meaning the trade-off between redundant computation and communication has to be factored in, so the tunable parameter values are more sensitive.

- **Problem Size Effect**: Finally we notice that increase in problem size leads to decrease in sensitivity, relative to smaller problem sized instances. This is observed from the increasingly flat bases of the violin plots when *dim* approaches 3100.

While these are high level observations, we now examine sensitivity effects in detail.

Detailed Examination of i7-2600K sensitivity We now explore interesting areas of the space in more detail by expanding segments of Figure 5.5, Figure 5.6 and Figure 5.7 into Figure 5.8. These are two samples of *dim*=[700, 2700] for *dsize*=[1, 5] representing the boundary problem and data sizes. They are followed by two samples of *dim*=[1100, 1900] for *dsize*=[3, 5] representing the mid range cases.

The samples in figures 5.8 (a) and 5.8 (b) are close to the boundary cases in our search space and they conclusively highlight how difference in problem size and data granularity (and corresponding variation in kernel task granularity within them) impacts the search space. For *dim*=700 we note that most of the points in *tsize*=100 to 1K are dispersed around the median value. This is due to the best configuration in these cases being all CPU (see the heatmap in figure 5.2 showing *band*=-1 for i7-2600K where *dim*=700, *tsize*$\leq$2K). In that case the tunable parameters are only *cpu-tile* and *dsize* resulting in configurations numbering in tens instead of thousands. Contrast this with *tsize*$\geq$2K and for all points in *dim*=2700 where there are many points less than the median value, as seen from the flat base of each violin. These cases correspond to various combinations of the tunable parameters *band*, *halo* and *gpu-tile* in addition to *cpu-tile*. We also observe that in case of *dim*=2700, *dsize*=5 variations in the former three parameters do not affect performance as much as for *dim*=700. This is also confirmed by the lower gap between average *rtime* and *ber* (see figure 5.4). However selecting the worst points in these cases, such as computing on the CPU only with *band*=-1 when *dim*=2700, *dsize*=1 and *tsize*$\geq$4K, is quite costly (up to 8 times slower). The worst case in these cases are the best points for *dim*=700, *tsize*$\leq$2K. Thus, while variation in tunable parameter values from the best values within a subset of input configurations may not affect performance, it can affect performance in other subsets.

The samples of figure 5.8 (c) and 5.8 (d) highlight the mid range case which highlight the low sensitivity of these points in contrast to samples of figure 5.8 (a) and 5.8 (b). The bases are flatter and there is hardly any difference between the violin plots for *dsize*=3 and *dsize*=5. While this may indicate tuning certain parts of the search space can be trivial,

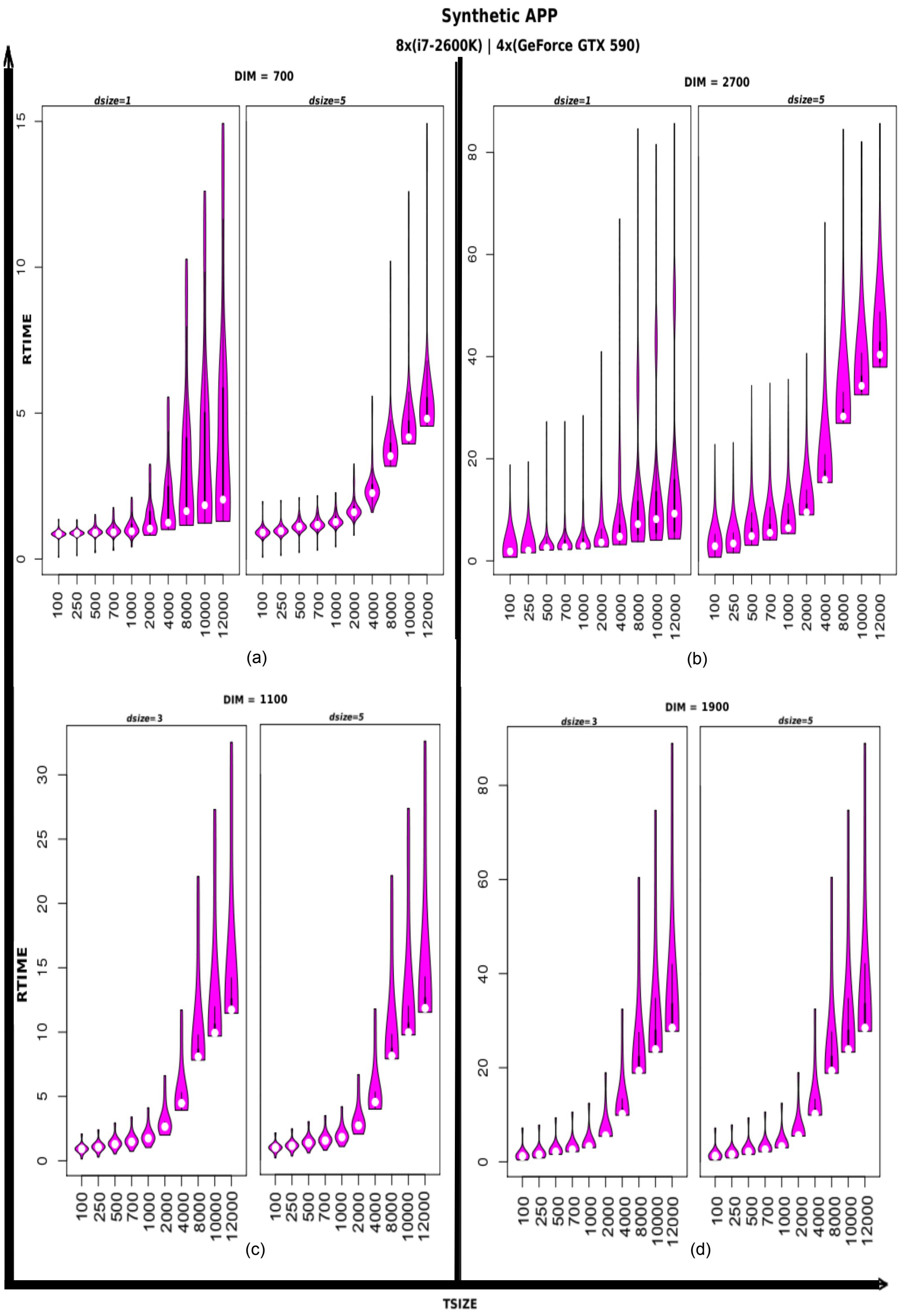


Figure 5.8: Detailed violin plots for the synthetic application showing dispersion of all configurations for the boundary cases of (a) [*dim*={700}, *dsize*={1,5}] (b) [*dim*={2700}, *dsize*={1,5}] and the mid range cases of (c) [*dim*={1100}, *dsize*={3,5}] (d) [*dim*={1900}, *dsize*={3,5}] in the i7-2600K system. The best points are at the base and the white spots are the medians. The x-axis is *tsize*, indicating kernel task granularity and the y-axis is actual execution time.

we note that the best points in some subsets were the worst ones in others and vice versa, meaning that any attempt to hand code heuristics for each case quickly becomes impractical.

i7-3820 Sensitivity We briefly discuss the sensitivity of the i7-3820 system for the synthetic application as illustrated in Figure 5.9, Figure 5.10 and Figure 5.11. It is similar to what was observed for the i7-2600K system barring minor differences due to difference in hardware. As observed earlier in the i7-2600K system, here too the sensitivity decreases with increase in both data granularity and problem size, while registering a slight increase when task granularity exceeds certain threshold values. The increase in sensitivity varies based on *dsize* due to the trade off from the communication cost of transferring and exchanging more data between the GPUs and the computation cost of each data point. For example in Figure 5.9, for *dim*=[2700, 3100] there is a marked ascent in the median value (denoted by white spot) from the base of the violin plots from *tsize*=2000 to 12000, and the best performing points can be 8 times faster than the worst performing ones. But in Figure 5.10, the ascent is not so steep, and the best performing points are 3-4 times faster than the worst performing points. This is because for low data sizes, the data transfer overhead is lower, and choosing the optimal GPU-specific tunable parameter values (*band* and *halo*) affects performance.

In all these figures, some of the violin plots do not capture the worst performing points because they exceed the time limit for computation set by us. As the worst performing points are not part of the training dataset, their exclusion does not affect tuning but the height of the violin plot in the search space is distorted. These points are (*dim*=1900, *tsize*=12K), (*dim*=2700, *tsize*=[8K, 10K, 12K]) and *dim*=3100, *tsize*=[4K, 8K, 10K, 12K]).

We conclude this subsection by noting how our exhaustive search results have revealed the difficulty in creating a naive heuristic to predict optimal values for tunable parameters. Thus, we pursue auto-tuning strategies based on machine learning.

5.3.2 Autotuning : The Learned Model

Our learned model is an M5 pruned decision tree, as discussed earlier in subsection 2.4.7. We did not use Linear Regression or SVM based universal tuners due to their poor performance in the previous chapter. A careful analysis of the search space of the synthetic application shows the best performing points in the synthetic application resemble a saw-tooth like pattern, progressively increasing in size with increase in *dim*, as illustrated in Figure 5.12. This sort of pattern fits quite well with a tree based learner, as can be observed from a fragment of the learned model predicting the optimum *halo* values for the i7-2600K system, in figure 5.13.

The regression equation (LM1) shows that *halo* depends on other tunable parameters like *band* and *cpu-tile*. This agrees with our intuition as *halo* values are a measure of the extent of overlap among partitioned diagonals offloaded onto GPUs. Hence, *halo* values depend on

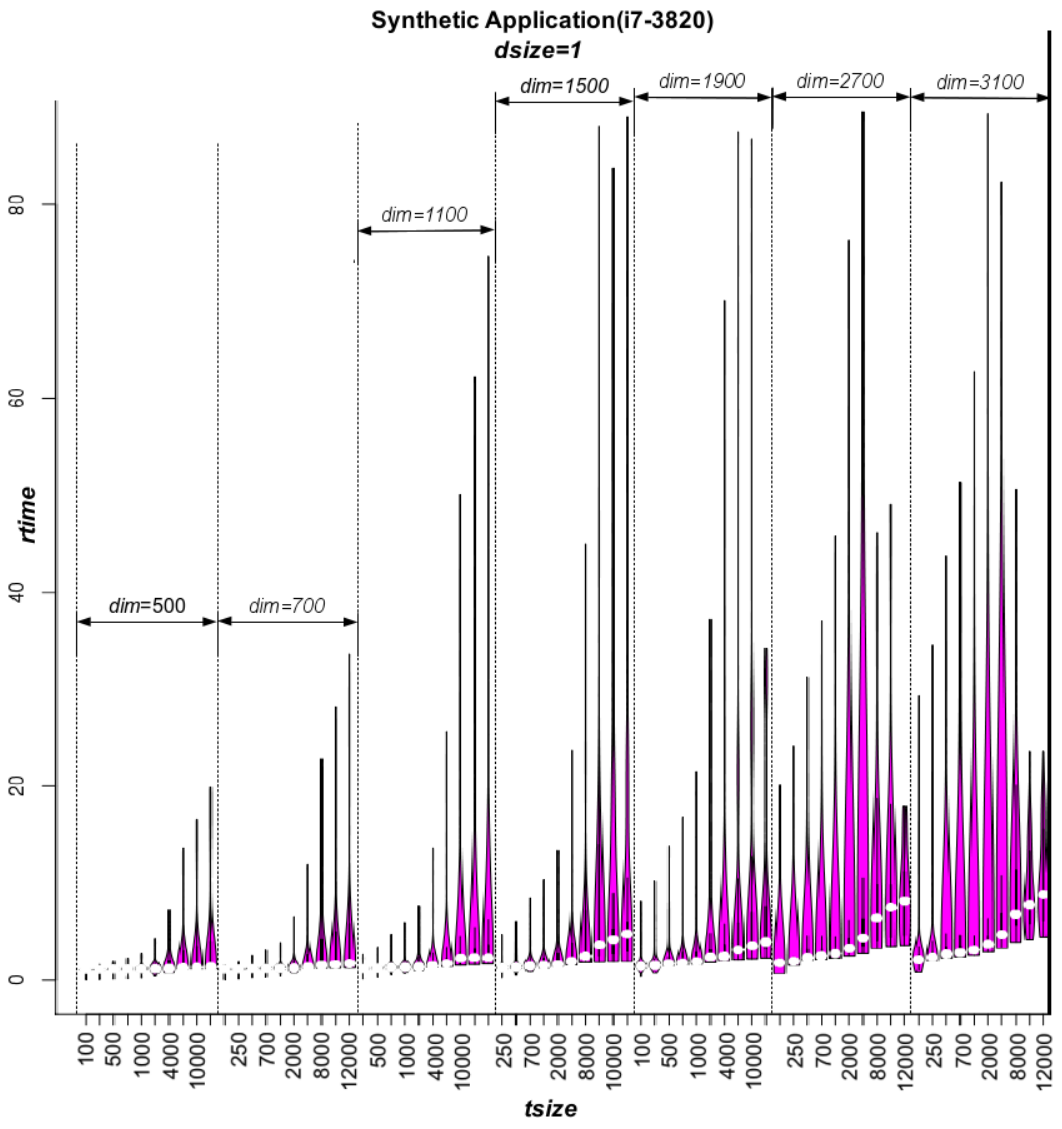


Figure 5.9: Violin plots showing dispersion of all configurations for the synthetic application in the i7-3820 system. The best points are at the base and the white spots are the medians. The x-axis is *tsize*, indicating kernel task granularity.The y-axis is *rtime*, indicating actual execution time. The violin plots cover the entire search space with *dim* varying from 500 to 3100, *tsize* from 10 to 12K and $dsize = 1$

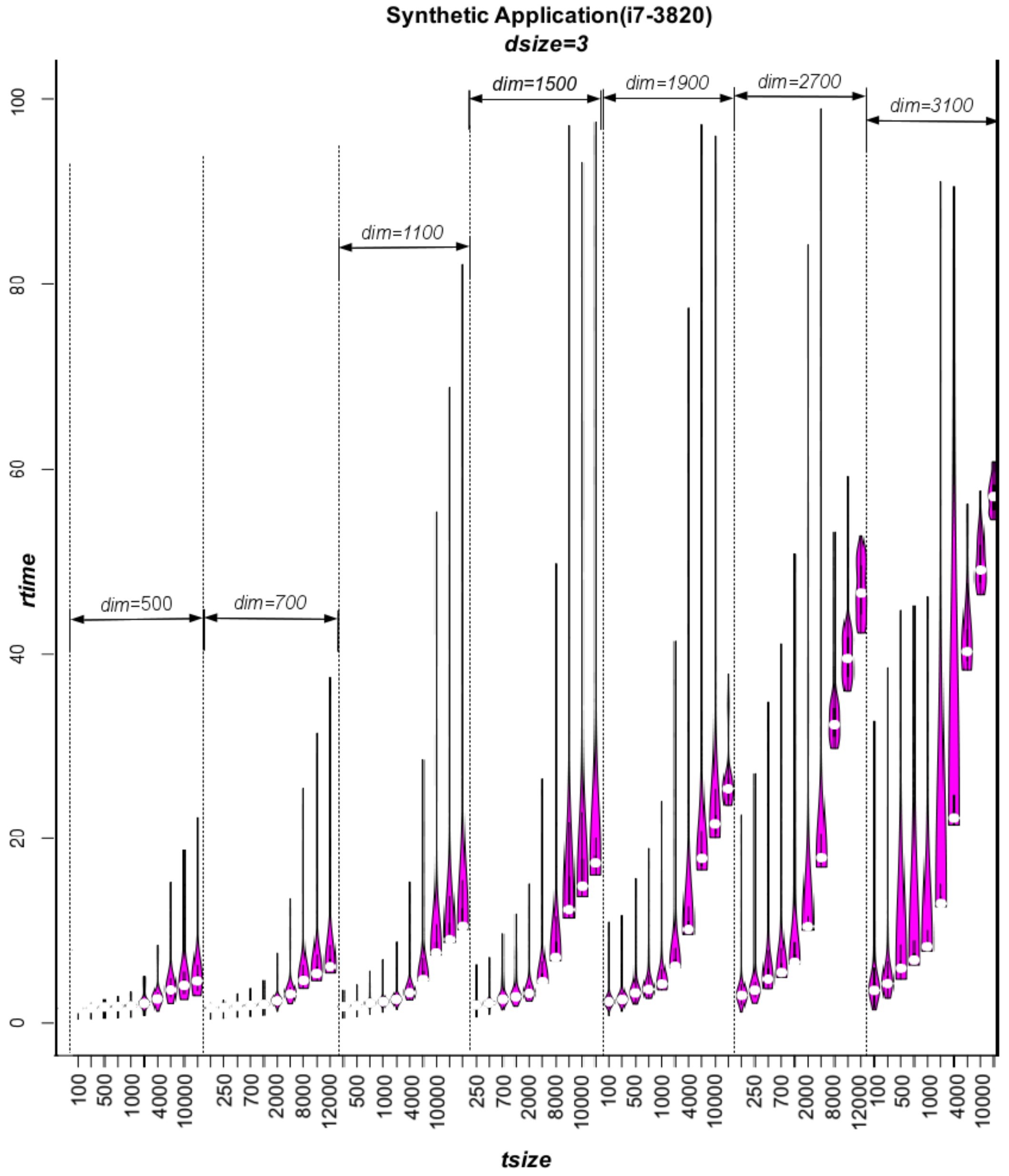


Figure 5.10: Violin plots showing dispersion of all configurations for the synthetic application in the i7-3820 system. The best points are at the base and the white spots are the medians. The x-axis is *tsize*, indicating kernel task granularity.The y-axis is *rtime*, indicating actual execution time. The violin plots cover the entire search space with *dim* varying from 500 to 3100, *tsize* from 10 to 12K and *dsize* = 3

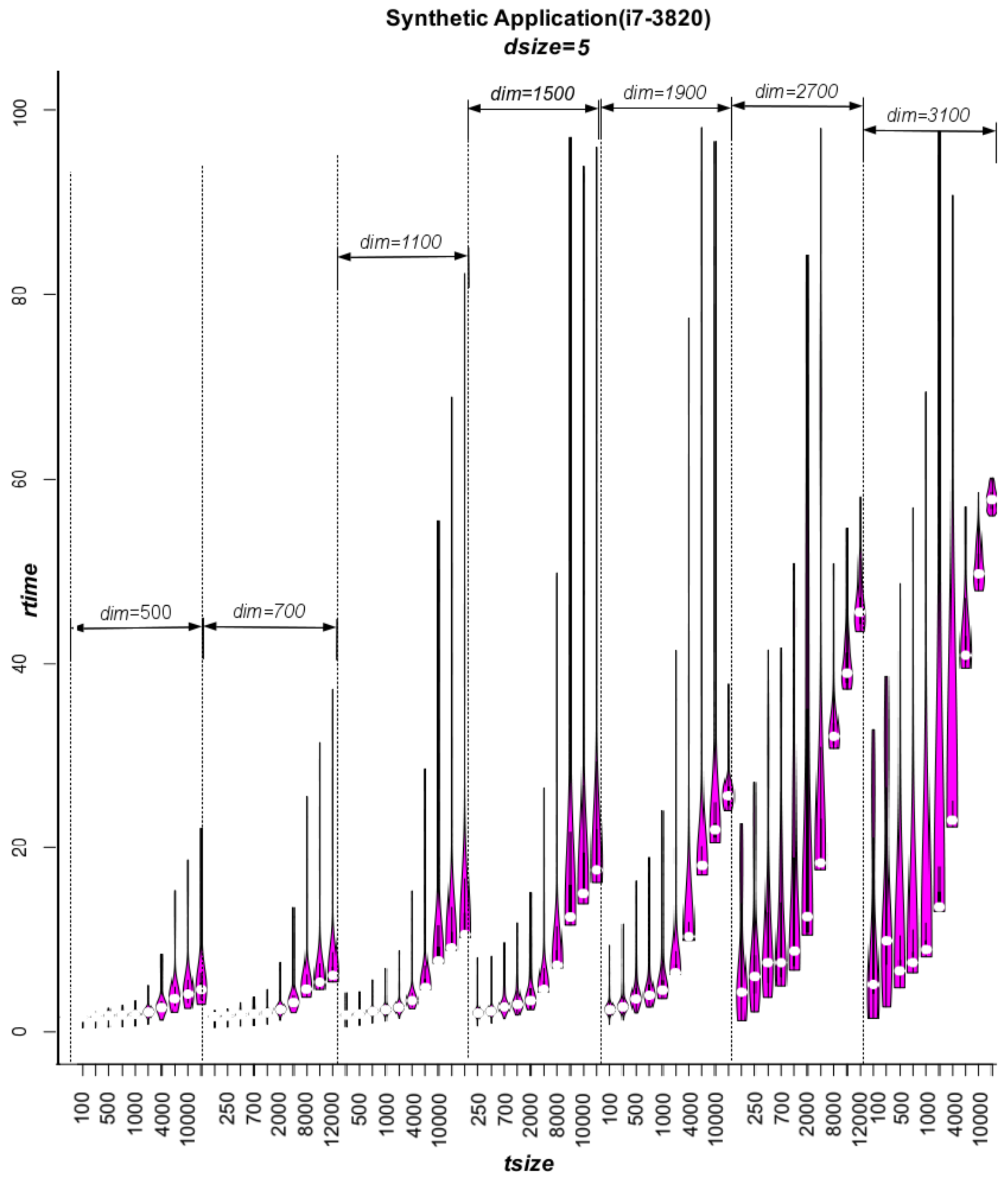


Figure 5.11: Violin plots showing dispersion of all configurations for the synthetic application in the i7-3820 system. The best points are at the base and the white spots are the medians. The x-axis is *tsize*, indicating kernel task granularity.The y-axis is *rtime*, indicating actual execution time. The violin plots cover the entire search space with *dim* varying from 500 to 3100, *tsize* from 10 to 12K and $dsize = 5$

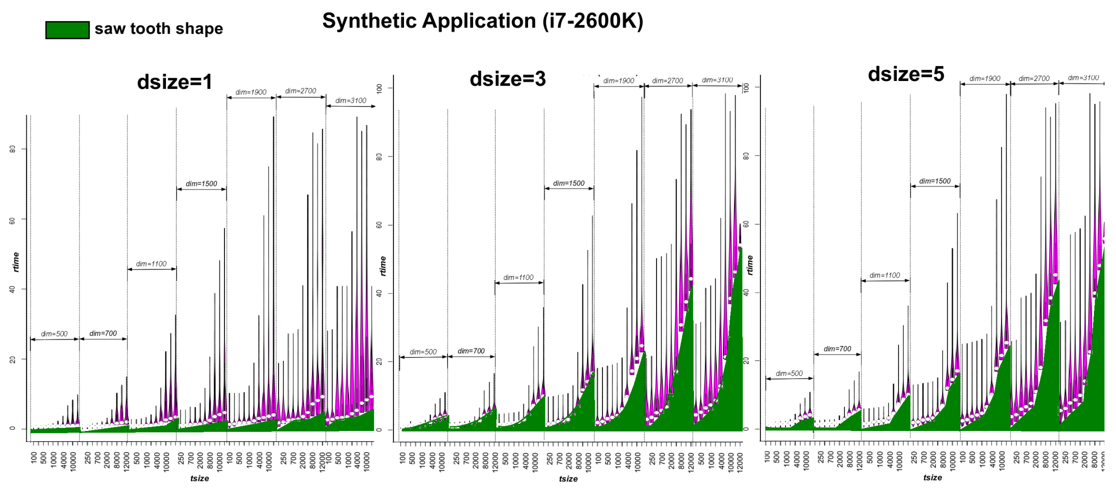


Figure 5.12: The saw tooth like shape of the best performing points in the synthetic application search space for i7-2600K system, with progressively increasing size as *dim* values increase. This shape is more pronounced with increase in *dsize*. This type of pattern is a good fit for tree based learners that generate different linear regression equations to predict output parameters based on decisions taken for different values of input features. An example of such tree is illustrated in Figure 5.13.

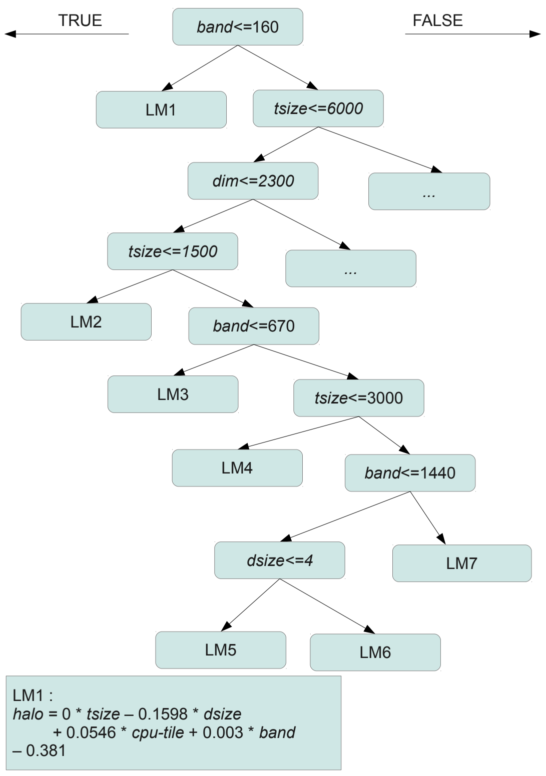


Figure 5.13: i7-2600K system : The M5 pruned model tree for predicting *halo* values with one linear model (out of 22) shown. As seen, *halo* depends on *band* and *cpu-tile* values, as well as the input parameters of task granularity and data granularity.

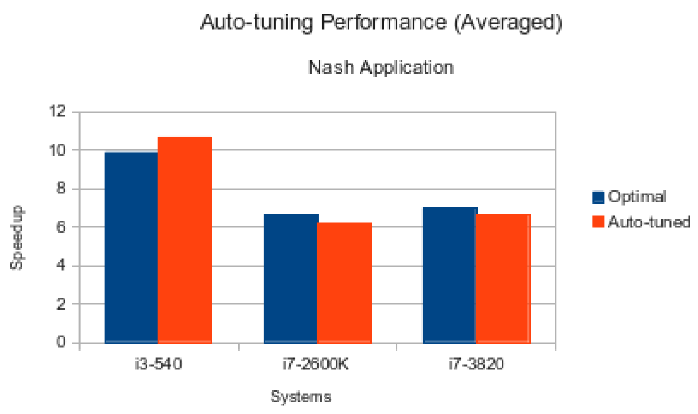


Figure 5.14: Speedup over sequential baseline from auto-tuning is within 5% of exhaustive search.

band values. *cpu-tile* values were predicted using input parameters only (*dim*, *tsize* and *dsize*). This was because on removing other tunable parameters from the regression equations that predicted *cpu-tile* values, accuracy of prediction increased. This also makes intuitive sense since an all CPU configuration has tiling as its only tunable parameter, so other tunable parameters are not needed. *band* values depended on *gpu-tile* values in addition to input parameters. From our exhaustive search we found *gpu-tile* values corresponded to either 1 or 0 (meaning a GPU was not employed), so it was a binary decision that was accurately predicted using REP Tree. *cpu-tile* and *band* values, like *halo* values, were predicted using the M5 pruned tree model.

5.3.3 Autotuning Results : Evaluation

For the fine grained Biological Sequence Comparison application, autotuning was trivial as the band predictions were 100% accurate, i.e. do everything on the CPU. Our learning model had predicted band=-1 for all $tsize<100$, across our search space of $dim\leq3100$. Thus in the context of our search space only the predicted *cpu-tile* values differed and selecting the best points was trivial.

A summary of our autotuner's performance for the Nash application is shown in figure 5.14. This figure describes for each system, the average optimal speed-up against a sequential baseline found during exhaustive search of Nash, and the speed-up obtained by our autotuner.

The super-optimal performance in the case of the i3-540 is explained by the fact that our decision tree based tuner is free to select parameter values which lie outside the set of cases explored in the necessarily finite search. The better quality predictions for the i3-540 can be explained by considering a) it is a single GPU system with only two tunable parameters *band* and *cpu-tile*, i.e. less parameter values to predict as compared to the multi-GPU systems and b) its four CPU cores are slow relative to its GPU, meaning most of the data is often

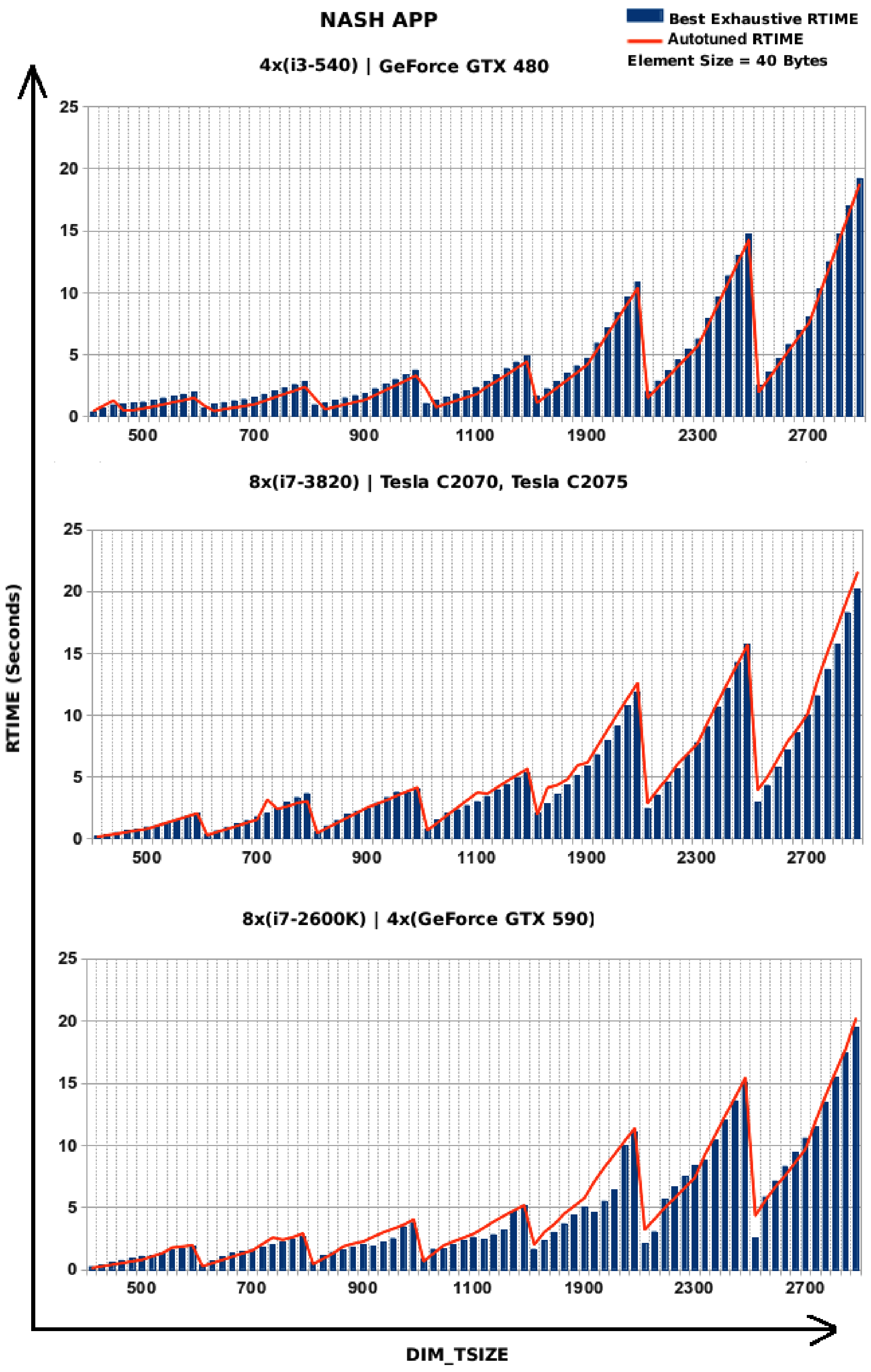


Figure 5.15: The bars represent runtime of optimal points found from exhaustive search and the line represents runtime from auto-tuning. The x-axis is *dim-tsize*, indicating groups of problem sizes whose task size varies from 10 to 12000 and y-axis is runtime.

offloaded onto the GPU, easing prediction as compared to the i7 systems with fast CPU cores.

We conclude this section with a detailed visualization of how our auto-tuning fares against the best exhaustive runtime or '*ber*' (figure 5.15). The *rtime* after autotuning is slightly lower than the *ber* for the i3-540 at many points (as discussed above), while it is slightly higher for the i7 systems as prediction is harder. Correct setting of the tuning factors leads to a maximum of 20 times speedup over an optimized sequential baseline, with an average of 7.8 times speedup. Our machine learned heuristics obtain 98% of this speed-up, averaged across all three systems.

5.4 Conclusion

In this chapter we discussed our autotuning strategy for 2D wavefront applications in systems that have two GPUs and the performance of such an autotuner. The introduction of an additional GPU enabled tasks to be split and offloaded onto each GPU, increasing scalability and performance. However, in such a setting the exchange of data in the boundary regions (*halo*) became a tunable parameter. Transferring less data was quicker but the number of transfers increased, leading to increase in communication costs. Similarly, the size of data (*dsize*) also affected tuning. We first explored the runtime performance of our synthetic application across a range of kernel task granularities. The search space of the synthetic application across three different systems for various combinations of input parameter values were explored using violin plots to analyze the sensitivity of the best performing points. We observed that well chosen settings for the *cpu-tile*, *band* and *halo* (for multi-GPU systems only) parameters produced significant speedups and correspondingly poorly chosen points were many times slower than the best performing points. Based on the shape of the search space, we decided to use decision trees to train on synthetic training sets, which were evaluated on real world applications. Our tree based learners were able to obtain on average 98% of the speedup of the best exhaustive runtime figures.

Chapter 6

Autotuning 3D Wavefronts

6.1 Introduction

In this chapter we describe our tuning strategy for 3D wavefront applications and analyze the results of our single GPU autotuning scheme. Our 3D tuner is an extension to the 2D autotuner described in the previous chapters.

We discuss our tuning strategy for the 3D case in section 6.2. This is followed by a discussion of our synthetic 3D wavefront training application and the real world lower triangulation application in section 6.3. In subsection 6.4.1 we discuss our findings from an exhaustive search of the best performing points for our synthetic application and use the lessons learned to train our autotuner. Our autotuning results on the real world application are discussed in subsection 6.4.2 and we conclude with a discussion of our multicore + single GPU tuning of 3D wavefronts in section 6.5.

6.2 3D Autotuning Strategy

The 3D case differs from the 2D case in that CPU tiling plays no significant role. This is because within the range of our experimental space spanning from 100 to 9,000,000 elements (*dim*=10 to 3000), for the same quantity of compute elements in the data grid, the cubical structure is already small enough in the i,j,k dimensions to take advantage of cache reuse compared to the matrix structure in the 2D case.

For example, in a 250×250 square matrix with 62,500 compute elements, tiling may be done with tile size of 10×10 to have 625 such tiles for cache reuse. However, as we have seen from our previous 2D results, tiling is not beneficial for the 250×250 case as the problem size is not large enough in the i,j dimension to overflow the cache. In our 3D case, the equivalent cubical datagrid is $40 \times 40 \times 40$ (we are ignoring $39.68 \times 39.68 \times 39.68 = 62500$ as problem size is limited to natural numbers) which is already small enough in the i,j,k dimensions to fit the L1 cache.

Similarly, even in the largest configuration of 3000×3000, the equivalent problem size in the 3D case would be $208 \times 208 \times 208$. In the i,j,k dimensions this is less than the *dim*=250 case where tiling provides no benefit. Thus unlike the earlier 2D cases, *cpu-tile* parameter is not needed in the 3D case.

With regard to training, we use the synthetic application to train our M5 pruned tree model based on a training set taken from our exhaustive search results in the same manner as the 2D cases. The training set comprises 5 best *band* values for each grouping of the input parameter values of *dim* and *tsize* from the synthetic application. Choosing a smaller number of training samples per input parameter configuration results in poor accuracy of prediction during k-fold cross validation of the synthetic application. Choosing a higher number does not improve accuracy either, due to the saturation limit of training examples for tree based learners, discussed in subsection 2.4.7. Our M5 pruned tree learner is tuned by checking the predictions against the test set. Since this is a single GPU system, *halo* is no longer a tunable parameter and we instead tune for the *band* values. We also do not tune for *gpu-tile* parameter. This is based on our observation of tuning results from previous chapters where *gpu-tile* values > 1 did not improve performance of 2D wavefront applications.

Our first tuning decision is a binary choice between using or not using the GPU in each system for different configurations of input parameters. We capture this information in a tunable parameter *useGPU*, and this binary decision is accurately predicted using a REP Tree. Apart from this trivial binary choice, the only tunable parameter for our single GPU-multicore tuner for 3D wavefronts is the *band*, which is predicted using the M5 pruned tree model. As before, a *band* value of -1 indicated no GPU usage. This may look like a redundant piece of information (as we already capture this in *useGPU*) but comparing *band* values of -1 with *useGPU* prediction values of -1 provides an additional validation for not using a GPU.

6.3 Application Suite and Platforms

In line with our previous experimental programs, we first build a synthetic 3D wavefront application to explore the search space and generate training data for our learning models. This synthetic application is identical to the 2D synthetic application except for the addition of the third dimension. The problem size and task size granularity of this application are controlled in the same manner as our 2D synthetic application. However, since our experiments are restricted to single GPU multicore architectures, our 3D application does not have the data size input parameter. This is because data size is only relevant when GPU to GPU boundary value swapping is required.

6.3.1 Lower Triangulation Application

We extracted our 3D wavefront application from a NAS Parallel Benchmark (NPB) program that deals with solving an unfactored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions using a technique called symmetric successive over-relaxation or SSOR[57]. A high level control flow for the SSOR subroutine is shown in the pseudo code listing below.

```
DEF SSOR
DO ISTEP=1,ITMAX
   CALL COMPUTE_RHS
   CALL JACLD
   CALL BLTS
   CALL JACU
   CALL BUTS
   CALL ADD
END DO
```

SSOR comprises multiple iterations until the solution converges to the desired value. The COMPUTE_RHS calculates pressure and viscous forces of the Navier-Stokes equation which are present on the right hand sides of the equation. Then the lower-triangular and diagonal systems are formed in the Jacobian Lower Diagonal method - (JACLD) and solved in the block lower triangular solution (BLTS). This is followed by the forming the strictly upper triangular part of the Jacobian matrix in the JACU subroutine, which is then solved by the block upper triangular solution (BUTS). It should be noted the solution at coordinates (i,j,k) depends on those at $(i+s, j, k)$, $(i, j+s, k)$ and $(i, j, k+s)$ where $s = -1$ for BLTS and $s = 1$ for BUTS, corresponding to two sweeps in one iteration. For a compute grid of size $nx \times ny \times nz$, the first sweep corresponds to the lower triangulation starting at $(2, 2, 2)$ and ending at $(nx-1, ny-1, nz-1)$, while the second sweep corresponds to the strictly upper triangulation algorithm flowing in the reverse direction. Finally the solution is updated.

Wavefront like behavior occurs within BLTS and BUTS methods as discussed in the subsection 2.6.3. However, the BUTS method is a backward sweep, so the planar waveflow shown in Figure 3.5 has to be in the reverse direction. Consequently, the dependency array in Figure 3.6 has to be modified. As our previous experiments dealt with forward sweeps, we limit our wavefront OpenCL kernel to the BLTS method, which also incorporates computational elements of the JACLD subroutine. This merged BLTS+JACLD OpenCL kernel has 530 lines of code, which is larger than all our previous 2D kernels put together. As before, we investigated a single sweep ($ITMAX = 1$) instead of multiple iterations that converge to a solution.

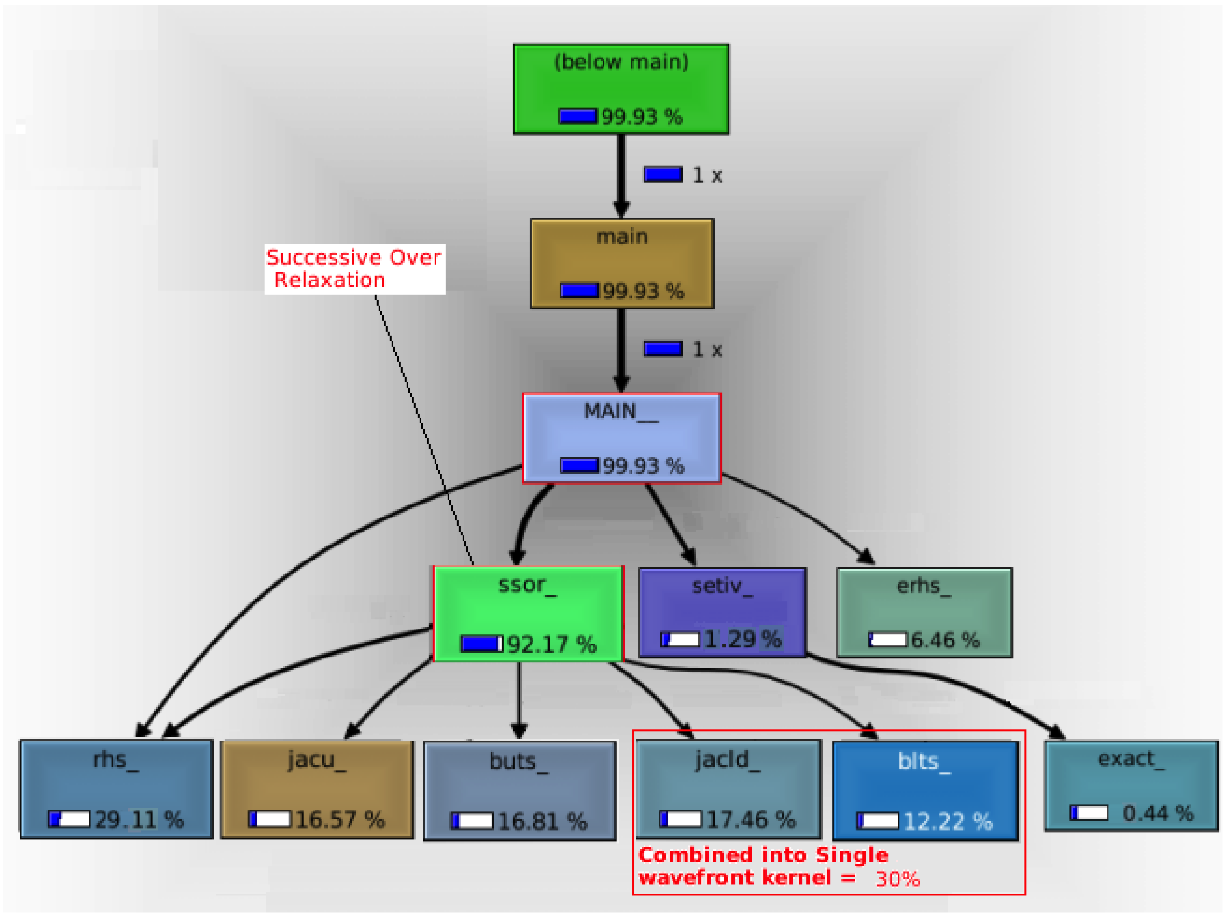


Figure 6.1: Call graph of the NAS Parallel benchmark application that solves a finite-difference discretization of the Navier-Stokes equation in 3D. It shows 92.17% of computation takes place inside the SSOR subroutine for a single iteration with two sweeps (lower and Upper). We have combined the Jacobi operation and the Lower Triangular operation into a single sweep of wavefront which accounts for 30% of overall computation

We verified the results from the original Fortran code against our optimized wavefront kernel code to ensure correctness of our merge optimization. The task granularity of our optimized single sweep **Lower Triangulation (LT)** kernel was found to be lower than that of the coarse grained Nash Application but larger than the fine grained Biological Sequence Comparison application from the 2D application suite.

The full Navier-Stokes NPB application also has a substantial residual computations outside the SSOR subroutine. This is shown in the call graph analysis of Figure 6.1 which illustrates the computation overhead for each of the subroutines

6.3.2 Experimental Platforms

To maintain consistency, our experiments are carried out on the same machines as for the 2D wavefront case. To recap, the i7 systems had fast CPUs and fast GPUs while the i3 system had a slow CPU and a moderately fast GPU. As before, we measure runtime of the whole

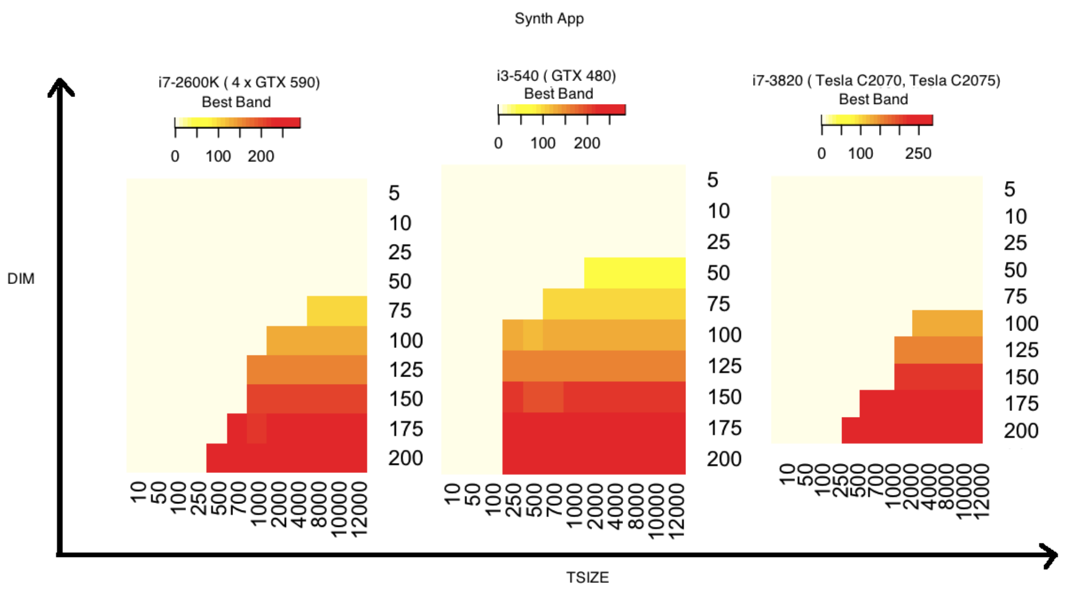


Figure 6.2: Exhaustive search results for the synthetic application on the i7-2600K, i3-540 and i7-3820 systems. The heatmaps illustrate the best *band*. In all maps the x-axis is *tsize*, indicating kernel task granularity and the y-axis is *dim*, indicating problem size.

program execution using wall clock timers in the host program, averaging across three runs. We also observed low variance between the runs which was in the order of few hundreds of milliseconds.

6.4 Results and Analysis

In subsection 6.4.1 we investigate the characteristics of the search space created by our 3D synthetic training application and the real world Lower Triangulation (LT) kernel, and explore the resulting models. In subsection 6.4.2 we evaluate our learning model on the real world kernel.

6.4.1 Exhaustive Search Analysis

We begin by examining the performance for the synthetic application in the search space across three systems in the form of heatmaps and violin plots. A set of three heatmaps corresponding to best *band* for various configurations of two input parameters (*dim* and *tsize*) across three systems are shown in Figure 6.2.

We analyze the best *band* results, which correspond to amount of GPU usage, based on the trade-off considerations discussed in section 3.4. In previous chapters, computing on the GPU ($band \geq 0$) became favorable when task granularity exceeded a certain threshold that varied with problem size and the hardware system. We make similar observations in the 3D case and they are as follows :

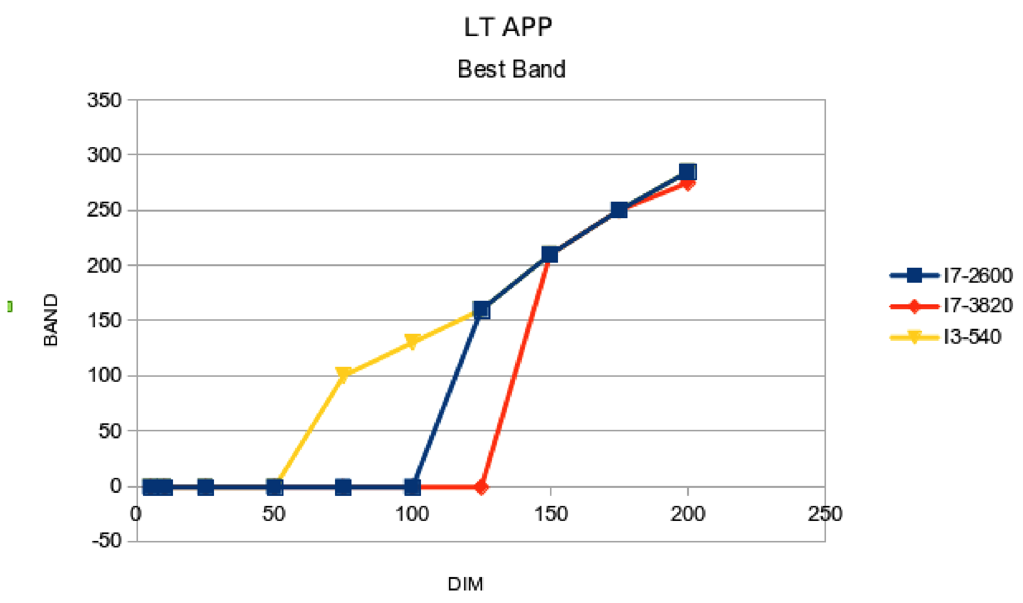


Figure 6.3: Exhaustive search results for the Lower Triangulation application on the i7-2600K, i7-3820 and i3-540 systems. The x-axis is *dim*, indicating problem size and the y-axis is the best *band*. There is no variation in *tsize* as the task granularity for this application is constant.

- **Problem Size and Task Granularity Effect** : Across all three heatmaps, GPU use is beneficial for problem sizes (*dim*) $\geq$ 150 and task sizes (*tsize*) $\geq$ 700. In general, the *band* value increases with increase in *dim* and *tsize*. This is because at higher task sizes the computation is coarse enough to overcome the communication overhead between CPU and GPU. Similarly at higher problem sizes, the number of concurrently computable elements in a plane diagonal increases, allowing the hundreds of GPU cores to dominate CPU only performance.

- **Variation between Systems** : As usual, the i3-540 system, has bands starting from a low *tsize* because its relatively slow CPU and 4 cores are easily dominated by the faster GPU with 15 compute units. Thus it makes sense to offload much of the data to the GPU in case of the i3-540, as is seen with the *band* ≥ 0 starting at *dim* ≥ 50, *tsize* ≥ 2000.

  In contrast, it is preferable to carry out a larger chunk of computation in the CPU for the i7-3820 system with its fast 8 core CPU as observed with the *band* ≥ 0 section, starting at *dim* ≥ 100, *tsize* ≥ 4000.

An exhaustive search for the best *band* on the LT application is shown in Figure 6.3. Since this application has a fixed *tsize*, *band* values vary with *dim* only. As with the synthetic case, the best *band* increases with increasing *dim* and *tsize*.

With respect to hardware, the trend is similar to the synthetic application where the i3-540 system starts offloading data to the GPU from *dim*$= 50$, owing to its slow CPU while

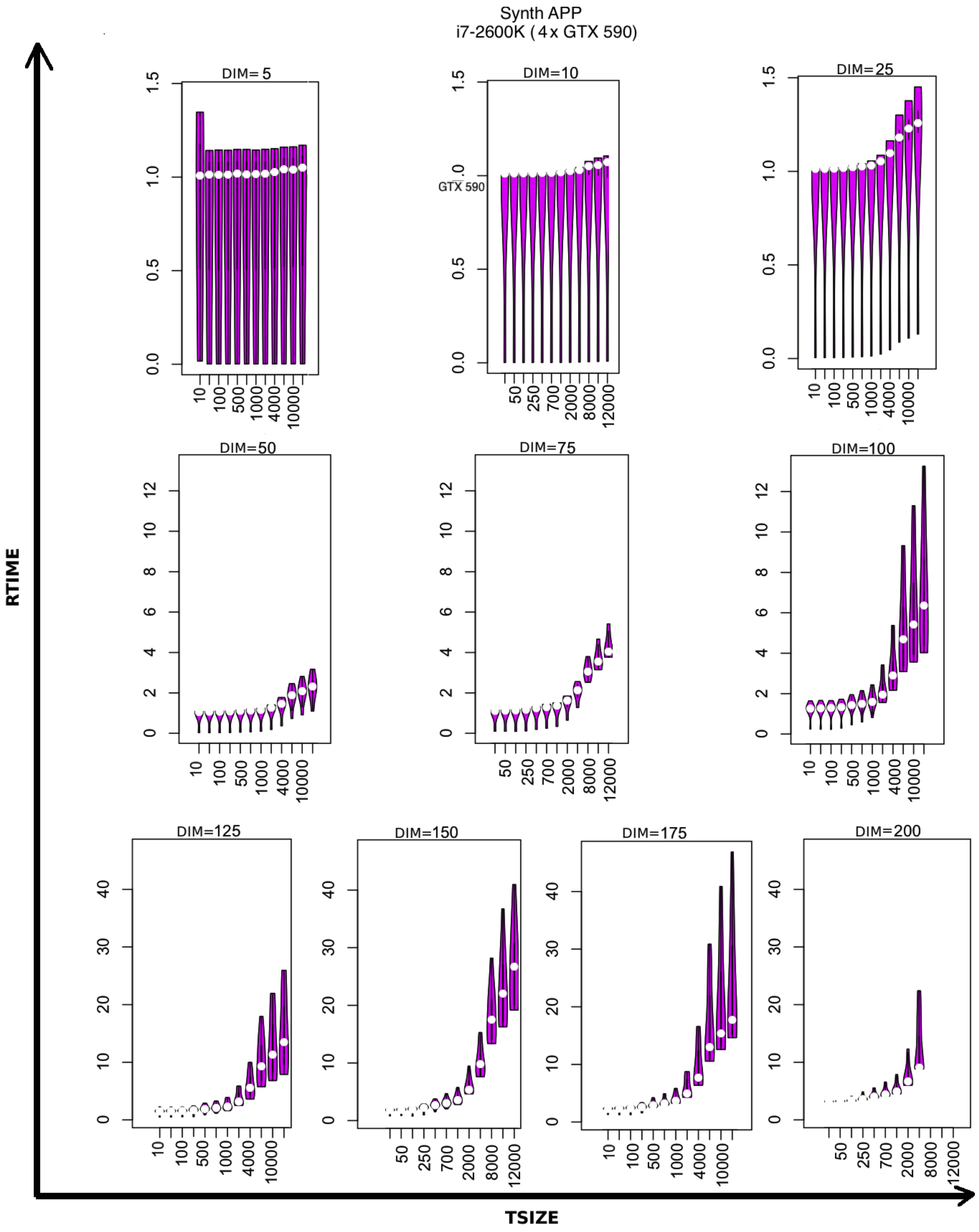


Figure 6.4: Violin plots showing dispersion of all configurations for the synthetic application in the i7-2600K system. The best points are at the base and the white spots are the medians. The x-axis is DIM_TSIZE, indicating a grouping of problem size and kernel task granularity. The y-axis is *rtime*, indicating actual execution time. The violin plots cover the entire search space with *dim* varying from 5 to 200, *tsize* from 10 to 12K.

the i7-3820 system, with its faster CPU, only offloads data from *dim*$= 125$.

6.4.1.1 Sensitivity analysis

We now explore the quality of the points at the best *band* values and their sensitivity to changes in parameter values. This is achieved through presentation of all points as violin plots. As discussed in subsubsection 5.3.1.4, the best performing points are at the bottom ends of the plots, with the median values being represented as white dots. Highly sensitive *band* configurations have narrow bases while flat bases mean we can choose from a wide range of *band* values and still achieve high performance, making prediction easy and accurate.

For each architecture, the violin plots have been grouped into three sections based on their *dim* for all values of *tsize*. These three groupings are 5-25, 50-100 and 125 to 200. The runtime at *dim*=5 is in the range of few hundreds of milliseconds while for *dim*=200, the runtime exceeds hundreds of seconds. Owing to this large disparity in runtime figures, the violin plots have been divided into the three groups for better visualization.

The sensitivity analysis of the i7-2600K system is shown in Figure 6.4. We make the following observations.

- For *dim*$= [5 - 25]$ the best point is hard to get as these points are highly sensitive. At such small problem sizes, the parallel CPU version with *band*$= -1$ is best, since there is not enough parallelism to cover the overhead of CPU to GPU transfers. All other bands fare poorly. Thus the band values are clustered around the median at the top end of the plot. For *dim*$= 25$, at a high *tsize*$> 8K$ the median approaches the middle of the violin plot. The best performing points are 3-4 times faster than the worst but in absolute terms they are mostly within *rtime*=1 second, so poor choice is not too costly.

- For *dim*$= [50 - 100]$ the *band* starts to matter for higher *tsize* across all dims. The median gradually comes to the middle and the base starts getting flat. Deviation from the best band has penalties, particularly for *dim*$= 100$ since the worst performing point at *tsize*$= 12000$ is three times slower than the best performing point at *rtime*=12 seconds.

- For *dim*$= [125 - 200]$ most bands above a certain threshold give good performance, so choice of band is not so sensitive within a broad range of good bands. However, making the wrong choice is now very costly as seen from *dim*=175 where the worst point is 3 times slower at *rtime*=45 seconds compared to the best performing point at *rtime*=15 seconds. It should be noted for *dim*=200, *tsize*$= [8K, 10K, 12K]$ the corresponding execution times exceeded the set threshold of 50 seconds, and were not captured.

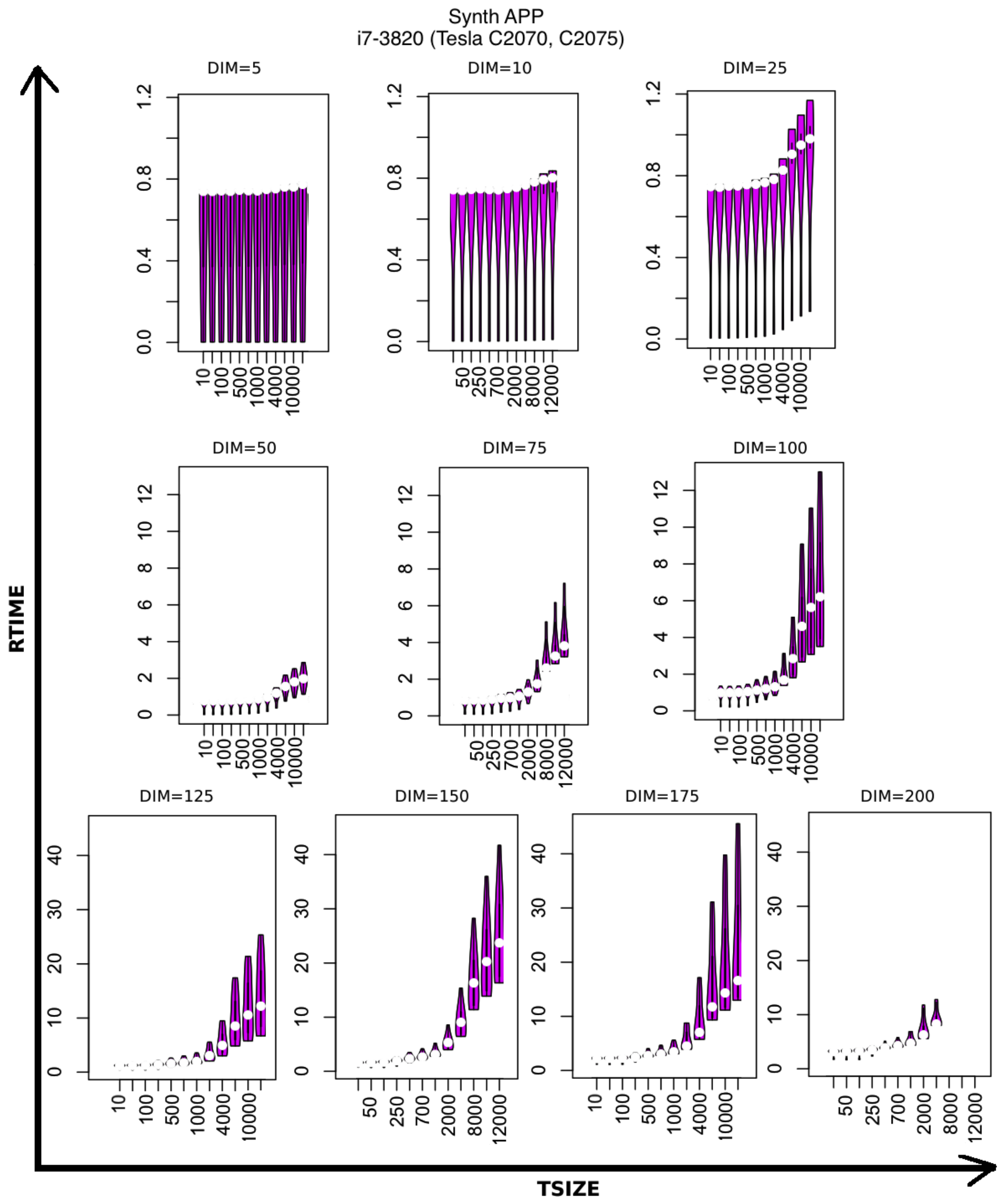


Figure 6.5: Violin plots showing dispersion of all configurations for the synthetic application in the i7-3820 system. The best points are at the base and the white spots are the medians. The x-axis is DIM_TSIZE, indicating a grouping of problem size, kernel task granularity and the data granularity. The y-axis is *rtime*, indicating actual execution time. The violin plots cover the entire search space with *dim* varying from 5 to 200, *tsize* from 10 to 12K.

The sensitivity analysis of the i7-3820 system is almost the same as for the i7-2600K system, with minor differences as shown in Figure 6.5.

However there is a marked difference for the i3-540, as shown in Figure 6.6, where the worst point is more than 15 times slower than the best point at *dim*=200, *tsize*=12000, due to the slow CPU. Thus carrying out the entire computation inside the CPU is quite costly for the i3-540 system. In terms of band values, the median value is usually near the flat base of most violin plots for this system. This means the i3-540 system is generally insensitive to choice of band as long as some amount of data is offloaded to the GPU. There are a few exceptions. In particular, for low *dim* of 5-25 where the median is at the top end of the violin plot, and for high *tsize* values for *dim*=100 where the median is slightly above the base.

We now discuss the sensitivity of the real world LT application, which is illustrated in Figure 6.7. In terms of hardware effects, there is a clear difference between the i3 and the i7 systems. Due to the fast GPU and slow CPU of an i3-540 system, offloading any amount of data onto the GPU is advantageous most of the time. So the sensitivity of the i3 system is low and this is reflected in its dispersion where the base tends to flatten earlier compared to the i7 systems. Moreover the worst points are more costly for the i3 system with the worst performing *rtime* of 12 seconds versus 5 seconds in the i7 systems.

We conclude this section by noting the large variations in best performing points across systems and applications, for different values of input parameters, make it hard to implement a naive heuristic that predicts the optimally performing points. Thus, as in previous chapters, we deploy machine learning based auto-tuning to find the best performing points. In the next section we explore the performance of our autotuner.

6.4.2 Autotuning Results

Once the parameters of our decision tree learner are tuned to predict with high accuracy for synthetic cases, we test our tuner on the LT application instances. The results of our autotuning are shown in Figure 6.9. We compare the autotuned runtime, shown as a line (Autotuned RTIME) over the best exhaustive runtime or *ber* (see subsection 4.3.2 where *ber* is explained). Except for a few instances like *dim*= 100 for the i7-2600K system and for *dim* values of 50 and 200 for the i3-540 system, the autotuner predicts with high accuracy.

On average, it even manages to predict super optimally (1.15% better than the *ber*) for the i3 system, in the same way it did for the 2D wavefront cases in subsection 5.3.3. Owing to the low sensitivity of the i3 system prediction is easy and accurate. Autotuning performs better than the *ber* because our tuner is free to select *band* values which lie outside the set of cases explored in the (necessarily finite) full search.

The autotuner also predicts marginally less well compared to the *ber* in the i7 systems on average, at (99.5%) in the i7-3820 system and at 96% for the i7-2600K system. A summary

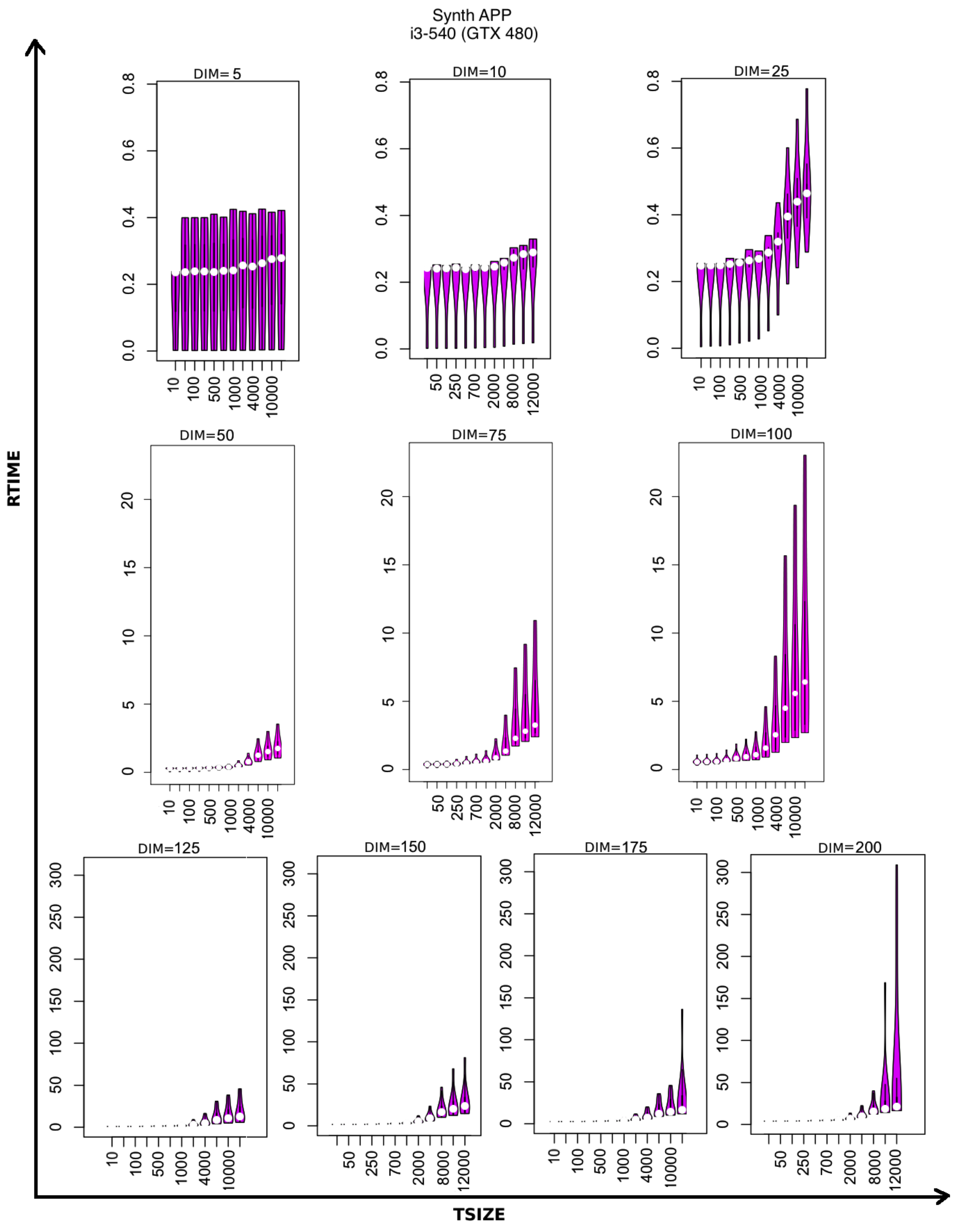


Figure 6.6: Violin plots showing dispersion of all configurations for the synthetic application in the i3-540 system. The best points are at the base and the white spots are the medians. The x-axis is DIM_TSIZE, indicating a grouping of problem size and kernel task granularity. The y-axis is *rtime*, indicating actual execution time. The violin plots cover the entire search space with *dim* varying from 5 to 200, *tsize* from 10 to 12K.

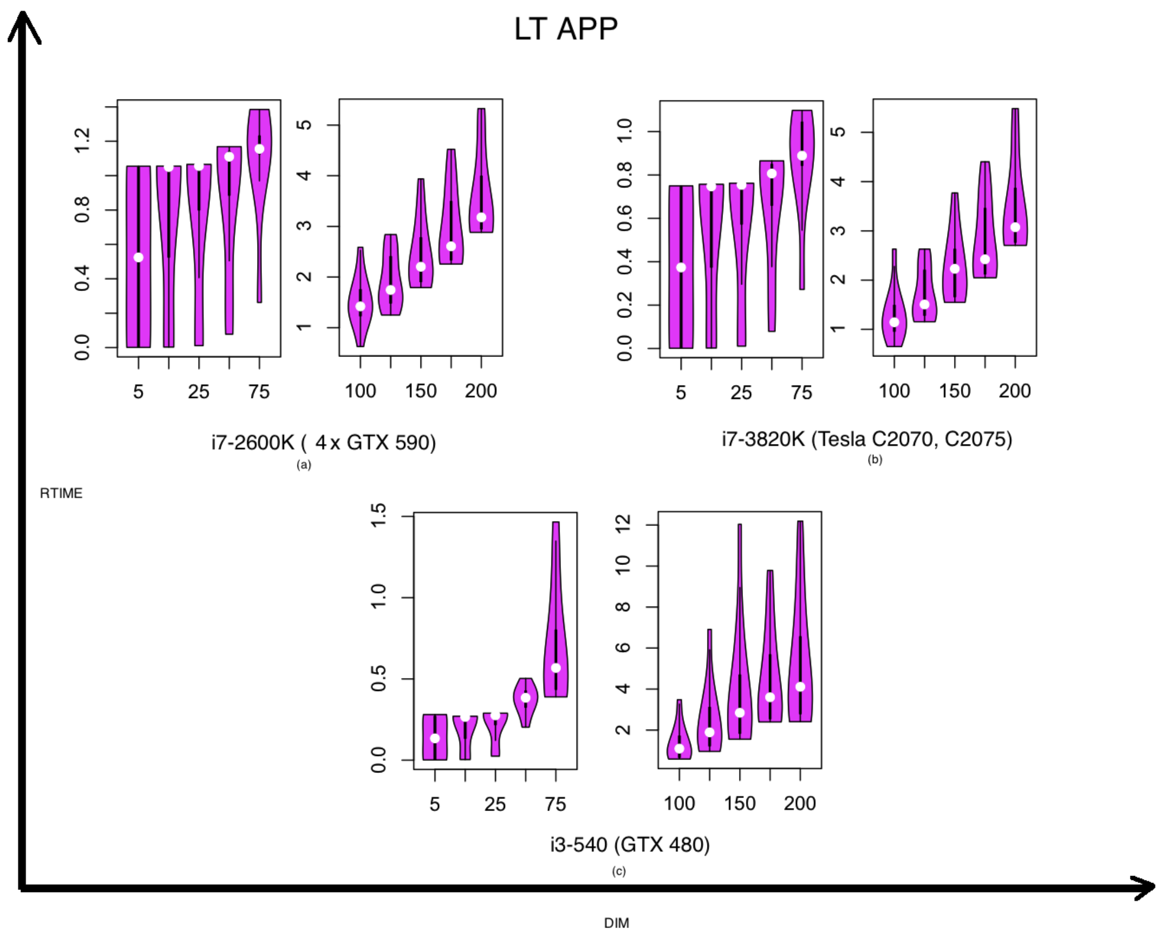


Figure 6.7: Violin plots showing dispersion of all configurations for the real world LT application across all systems. The best points are at the base and the white spots are the medians. The x-axis is DIM, indicating problem size. The y-axis is *rtime*, indicating actual execution time. The violin plots cover the entire search space with *dim* varying from 5 to 200.

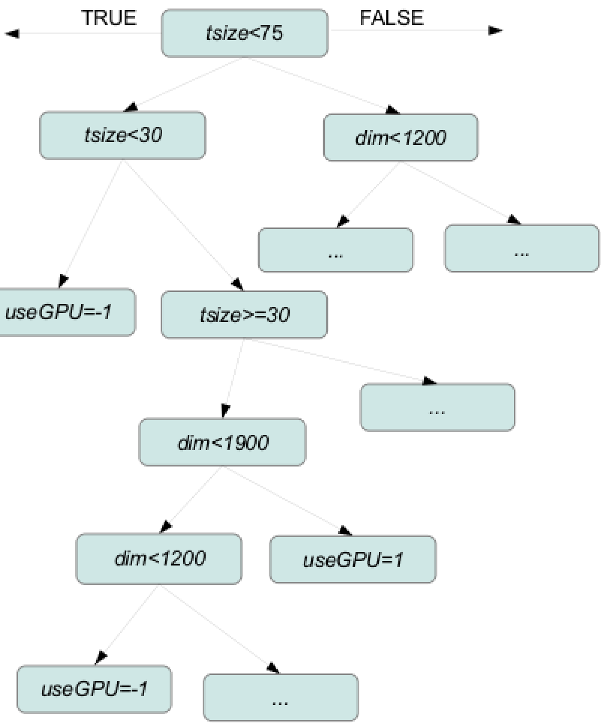


Figure 6.8: The REP tree for predicting binary *useGPU* values,i.e. 1 indicates GPU usage while -1 indicates no GPU usage. As seen, *tile* depends on the input parameters of task granularity and data granularity.

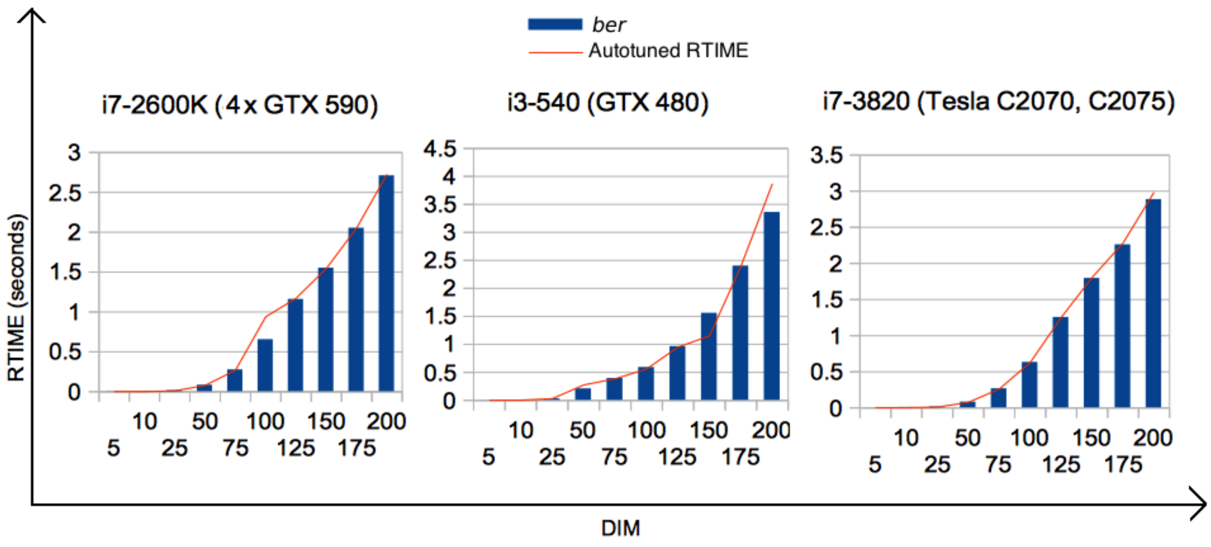


Figure 6.9: Plot showing best exhaustive runtime (*ber*) as bars and autotuned *rtime* as lines across three systems for the LT app. The x-axis is DIM, indicating problem size. The y-axis is *rtime*.

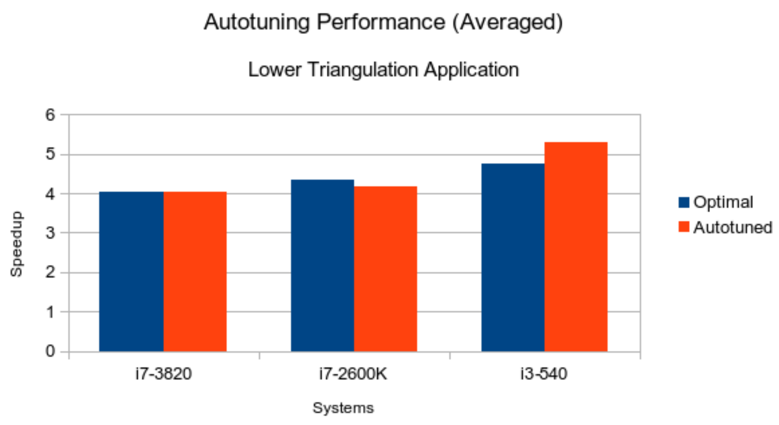


Figure 6.10: The bars represent averaged speedup of optimal points found from exhaustive search over the sequential baseline and the speedup from auto-tuning. The x-axis represents the systems and y-axis is speedup.

of this performance is illustrated in Figure 6.10

6.5 Conclusion

In this chapter we discussed our implementation strategy for tuning 3D wavefront applications. The addition of an extra dimension to a 2D compute grid made the use of *cpu-tiling* redundant as for the same amount of data, the size of compute grid in each of (i,j,k) dimensions is shorter compared to (i,j) dimensions, such as $4 \times 4 \times 4$ versus 8×8. This form of built-in tiling led to reuse of CPU caches. We created a 3D synthetic wavefront application to build our automated tuner and tested it against the real world Lower Triangulation application. Our tuner, built on a decision tree learner, was highly accurate.

Chapter 7

Related Work

7.1 Introduction

This chapter discusses prior work related to the areas introduced in the previous chapters. A comprehensive review of publications in each area is provided. To structure this discussion we follow the categorization of Williams [102], which notes that the optimization space, code generation, and exploration form the basic concepts of auto-tuning. This involves enumerating a large optimization space, generating code for those optimized kernels and finally, exploring the optimization space by benchmarking some or all of the generated kernels and searching for the best performing implementation. It categorizes autotuners into the following :

- Self-tuning library generators that target specific computation kernels or a class of kernels;
- Compiler-based autotuners that automatically generate and search a set of alternative implementations of a computation;
- Application-level autotuners that automate empirical search across a set of parameter values proposed by the application programmer.

We review autotuners that fall in one or more of these three categories, starting with the work done in the space of high level domain specific auto-tuning in section 7.2. We contrast the differences between automated tuning at low level with our own domain specific tuning of wavefront parameters at the high level of a skeleton. Since not all domain specific auto tuners utilize machine learning, we discuss prior work done in learner based auto tuning, particularly in low level compiler search space exploration and optimization, and contrast those with compiler based autotuners that do not use machine learning in section 7.3. Finally, all of the tuners discussed so far, including our own, use offline tuning. So we conclude the chapter with a contrasting discussion of online autotuning strategies in section 7.4.

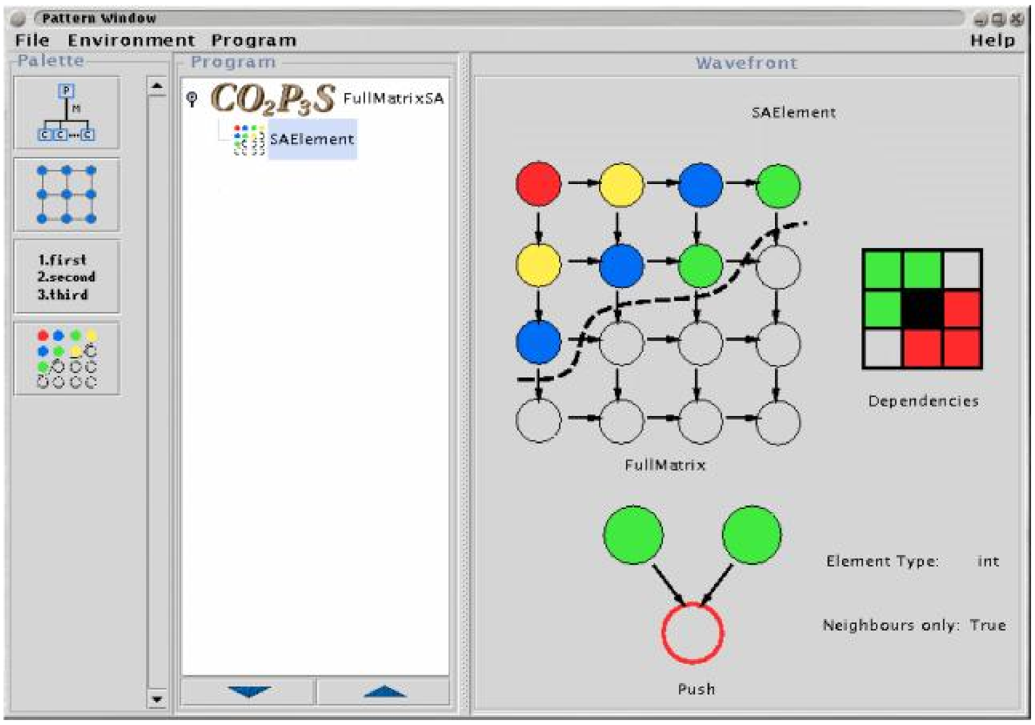


Figure 7.1: A GUI to design wavefront structure with Full Matrix shape parameter, {N,W,NW} dependency parameter and immediate neighbor only performance parameter, adopted from [5]

7.2 Domain Specific Autotuners

We begin with a discussion of parallel programming systems that generate structural framework code for pattern description of wavefront problems.

7.2.1 Wavefront Tuning Frameworks

CO_2P_3S [5] is a wavefront framework that generates parallel programs from user supplied methods and data. Its wavefront pattern template has design parameters that affect the parallel structure and performance parameters that affect code runtime. The design parameter shape "Full Matrix" enables computation of all elements in a rectangular matrix, "Triangular" supports computations over half of a matrix and "Banded" computes elements centered around the main diagonal of the matrix. The dependency set design parameter specifies the neighboring elements such as N, W, NW, meaning each element is computed by using previously computed values of north, west and north-west elements. For computing each element, CO_2P_3S provides users with a performance parameter called "neighbours-only". This parameter allows access to all previously computed neighboring elements when set to false, and access to only elements in the immediate neighborhood when set to true. A sample CO_2P_3S framework wavefront design is illustrated in Figure 7.1

While our wavefront framework offers similar features, including allowing users to specify dependencies (north, west, north-west), it does not provide access to all neighboring dependencies to compute an element. However, CO_2P_3S is restricted to just shared memory architectures and does not employ any optimization techniques for any combination of its application dependent properties. It also does not support any 3D wavefront problems (there is no *top* element dependency).

The wavefront abstraction in [104] targets multicore and distributed systems. However, its tunable parameters are specific to distributed systems and there is no support for 3D wavefront applications. It also employs processes instead of threads as they are more adaptable to distributed systems. The overhead from processes impacts performance in shared memory systems as opposed to light weight threading building blocks used in our work. There is also no exhaustive search space exploration and tuning using learning involved.

While these frameworks have not explored any optimization space, extensive work has been done on building performance models for wavefront applications by the Warwick High Performance Systems Group in [79, 50]. The performance models include an analytic "Plug-and-Play" reusable wavefront model on regular orthogonal grids, designed from performance studies of 3D wavefront applications like the Sweep3D and Chimaera particle transport benchmarks and the NAS-LU computational fluid dynamics benchmark. This reusable model targets large scale MPI based traditional HPC systems consisting of clusters of nodes numbering up to 8192 processors. It captures application level 3D wavefront parameters such as input problem size, number of sweeps, behavior of a sweep, height of tile, message size etc and machine specific parameters like "grind-time" per grid-point or (Wg), representing the non-idle time a given processor and compute time on each processor prior to communication (W). It utilizes information about these parameters to predict the runtime and scaling behavior of such MPI-based pipelined wavefront applications which help in hardware procurement and in optimizing application and hardware specific parameters.

The Warwick High Performance Systems Group has also explored two tier heterogeneous parallelism comprising clusters of CPU + single GPU systems in [82] for pipelined wavefront computation of the three dimensional LU application using MPI-CUDA. In this scheme a k-blocking strategy is employed to avoid idle work-items on the GPU whenever the size of the diagonal hyperplane (grid points with coordinates $i + j + k = w$ for w^{th} diagonal) is less than the number of work-items the GPU can execute in parallel. Other optimizations include loop unrolling and fusion, memory access coalescing for GPU and message size optimization when MPI is used for communication between processing-elements. Their previously built performance models are then used to study weak and strong scaling behavior of the code and investigate the overheads from computation, network communications and PCIe transfers.

This work is later extended to assess the performance portability of OpenCL based single GPU implementation across a cluster of nodes in [83]. Auto-tuning is done on two important

GPU parameters of memory layout and work-item distribution to improve performance without changing the underlying algorithm, apart from extending the k-blocking optimization for CUDA devices to OpenCL. For coalesced memory layout optimization, the choice is between an array of structs (AoS) approach that provides good cache utilization for processing a scalar work-item and struct of arrays (SoA) approach that allows parallel SIMD processing. For work-item and work-group distribution the choice is between fine grained distribution of one work-item per LU grid-point and coarse-grained distribution of one work-group per the GPU compute unit. However, this is not a machine learning based auto tuning framework.

With this discussion of work done on wavefront tuning frameworks, we look at other domain specific tuners ranging from highly specific library generators to the more generic stencil pattern autotuners.

7.2.2 Self Tuning Library Generators

ATLAS [58] is a self tuning library based on compile time auto-tuning of basic linear algebra subroutines. It includes some kernels from the Linear Algebra Package. It uses an exhaustive orthogonal search for empirical optimization. An orthogonal search refers to search in one dimension at a time by keeping rest of the parameters fixed. Architecture specific knowledge provides the possible range of optimizations which includes information like size of caches, number of floating point units, hardware characteristics, length of pipeline, natively supported instruction set and other low level details. This corresponds to ad-hoc empirical studies carried out for compiler optimization which involve changing flags for performance-critical sections of code.

Other tuning library generators are FFTW [39], OSKI [100] and SPIRAL [84]. FFTW is an efficient compile time autotuner that combines static models with empirical search to optimize FFTs while OSKI has both compile and runtime auto-tuning capabilities for tuning sparse linear algebra kernels. Unlike the previous three library generators that relied on simple string manipulation and were limited in scope to specific kernels and hardware, SPIRAL is a more generic domain specific auto tuner. It targets a class of kernels by generating empirically tuned Digital Signal Processing (DSP) libraries. SPIRAL and FFTW do not employ exhaustive search like ATLAS or OSKI due to their much larger optimization space relative to the other library generators. So they traverse their search space by employing search heuristics that include genetic algorithms.

Our experiments are quite similar with respect to carrying out empirical optimization for choosing the values of various tunable parameters. The crucial differences lie in our focus on exploring the optimization space of a high level abstraction such as the wavefront that is not tied to specific algorithms, kernels or subroutines (making it closer to SPIRAL) and in employing machine learning to tune the kernels for different computing systems.

```
void SimpleFarm1for1 (eSkel_atom_t *worker (eSkel_atom_t *),
  void *in, int inlen, int inmul, spread_t inspr, MPI_Datatype
  inty,
  void  *out,  int  outlen,  int  *outmul,  spread_t  outspr,
  MPI_Datatype outty, int outbuffsz, MPI_Comm comm);
```

Figure 7.2: Basic Farm prototype in *eSkel*, adopted from [73]

However, improvements to these library generators have been explored by employing machine learning in [67], where the best sorting algorithm is encoded as a function of the entropy of the input data and chosen at runtime for each target machine. We discuss this further in section 7.4. Other autotuners like William's [102] determine the best combination of low level implementation and data structure pertaining to the architecture and input data for two computational *motifs* (as per the Berkeley View terminology [6]) - computational grids and sparse linear algebra.

7.2.3 Task Farm Skeleton Frameworks

Another high level pattern or skeleton is the Task Farm which was briefly discussed in subsection 2.3.1. An example of the basic farm implemented in the eSkel library [73] is the Farm1for1 skeleton. The programmer specifies the number of workers, the operations that need to be performed by the worker in a task and the total collection of such input tasks. The skeleton handles the entire low level implementation ranging from getting the initial tasks, enforcing good load balancing through dynamic task distribution and collecting the results returned. In this classic farm implementation the farmer is also a multi threaded worker process. Each worker is created in the context of a dynamically changing hierarchy of MPI communicators so that its processes can communicate within the safety of that context. In case thread safe MPI is not available, each worker has to be a single process. Its prototype is shown in Figure 7.2.

The first parameter denotes the worker function which contains the code for solving a specific problem. Other input parameters include number of tasks, task element type, an output buffer that detects any overflow and the communicator within which the farm is constructed. A similar task farm library is the Munster Skeleton Library (Muesli) [62], a C++ skeleton library with an object oriented implementation. Compared to eSkel which exposed a low level API based on C binding to MPI, Muesli is based on the two tier model of P^3L [7] that also incorporates data parallel skeletons.

Here the client supplied method (foo()), shown in line 2 of Figure 7.3 is passed to the worker atomic process and then the worker processes are passed to the Farm. The Pipe in line 5 consists of three stages with initial, farm and final processes in each stage. J. Falcou et al have found in [33] that Muesli satisfies the principles of minimal conceptual disruption due to its approach of polymorphic C++ skeleton library which is familiar to

```
// step 1: create a process topology (using C++ constructors)
  Initial<int>        p1(init);
  Atomic<int,int>     p2(foo,1);
  Farm<int,int>       p3(p2,numworkers);
  Final<int>          p4(fin);
  Pipe                p5(p1,p3,p4);

// step 2: start the system of processes
  p5.start();
```

Figure 7.3: Task Farm in *Muesli*, adopted from [62]

many developers. Also, it accommodates diversity owing to its polymorphic nature. However when it comes to payback, the price of overhead is high [63]. Also, limitations of skeletons in Muesli cannot be easily circumvented using ad-hoc parallelism as was done in eSkel. Neither farm frameworks employ any form of sophisticated auto-tuning strategy and can at best be considered skeletons for the task farm pattern.

7.2.4 Stencil Tuning Frameworks

Stencils are high level patterns, but have a different dependency pattern compared to wavefront. Stencil computations are nearest neighbor computations, which are at the heart of structured grids and PDE solvers. Stencil application domains include heat diffusion, climate science, electro magnetics and fluid dynamics and their implementation in multi-GPU systems has been explored in [97]. Unlike wavefronts, all elements in the computation grid can be computed simultaneously and per element computation usually depends on neighboring north, east, west and south elements for 2D stencil applications like the Jacobi application [65]. In this section we discuss in depth the Berkeley autotuner proposed by Kamil et al [60, 59], while briefly discussing PATUS [18] which is an enhancement to the Berkeley design of Kamil et al .

The Berkeley auto tuner is a domain specific tuner that optimizes stencil computations across a range of CPU cache based architectures and NVIDIA CUDA based GPU architectures. Its stencil benchmark kernels consist of finite difference methods applied to Laplacian, Divergence and Gradient differential operators. Its code-generation framework consists of parallel and serial components and the tuner features an automated search that replaces stencil kernels with optimized versions. The input to the autotuner is a sequential Fortran 95 stencil expression and the final output is a tuned parallel implementation in Fortran, C or CUDA. This framework offers performance portability across chip multiprocessors and accelerators in the shared memory space but does not address distributed clusters. The basic mechanics of the framework are :

- User described problem specification in Fortran is parsed into an intermediate abstract syntax tree (AST) stencil representation.
- Numerous CPU or GPU specific transformations are explored utilizing the intermediate

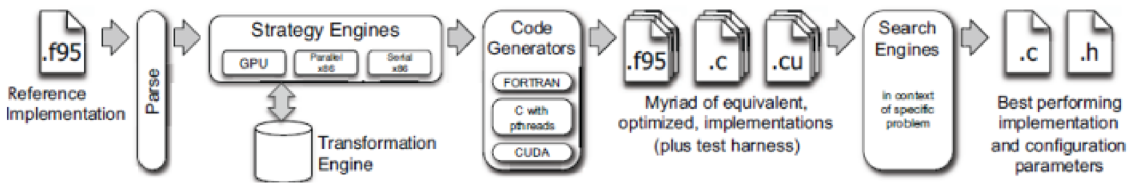


Figure 7.4: Berkeley Tuner flow showing parsing of readable domain specific code into abstract code by the parser. This is followed by transformation of code based on strategy for GPU, multicore or serial systems and generating equivalent Fortran for serial, C with pthreads for multicore and CUDA for GPU systems. Finally the search engine picks the best performing implementation [59]

representation.

- Optimized shared-memory parallel (SMP) implementations are generated and the best performing optimization and configuration parameters are determined via an automated search engine.

Figure 7.4 describes the flow of the tuning framework and we briefly discuss the main stages.

Front End Parsing Stage : Input stencil code in Fortran is parsed into an AST representation by the front end parser for subsequent transformation in later stages.

Strategy Engines Stage : A subset of the parameter search space is traversed for each architecture based on the best utilization of the underlying hardware such as employing cache blocking in the unit stride dimension for the Victoria Falls architecture [23], while ignoring the same for the Nehalem architecture [64], due to presence of hardware prefetchers. The engines also ensures parameter values are valid for the specific architecture, such as fixing the correct maximum number of threads based on hardware restrictions.

Code Generation Stage: This stage deals with production of code by the serial and parallel code generating components based on parameter values chosen in the strategy engines stage. Posix threads are used for parallelization in CPU cache-based systems of Nehalem and Victoria Falls while CUDA threads are executed on NVIDIA GPUs. In CPU systems the problem space is decomposed into three nested levels starting from the top level core blocks (size tuned to avoid capacity misses in the last level cache) which are decomposed into thread blocks sharing a common cache. These thread blocks are further decomposed into register blocks through strip mining and loop unrolling of stencil loops via the serial code generator. In GPU systems the CUDA code generator divides the problem across thread blocks and explores setting of the number of threads in a thread block and their access patterns in the global memory. Since parallelization is architecture specific (CPU or GPU), the parallel code generators modify the AST accordingly and pass the modified code over to the serial component.

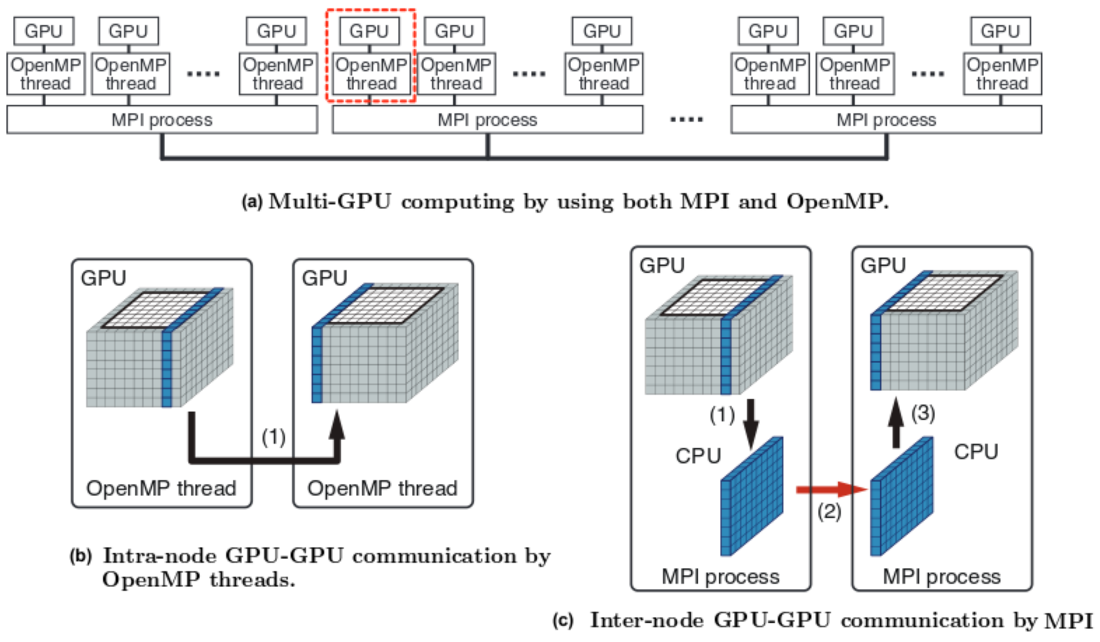
Search Stage : In this stage the autotuner records the runtime for each generated code on a target architecture and reports it to the users, who can then link the optimized stencil code into their existing code. The parameter subspace search is static and timing feedback is not used to dynamically direct the search.

The tuning framework reported the best speedup of 22x over the reference serial version and the performance was comparable to previous hand-optimized code at a fraction of the effort (minutes versus months).

Our framework does not generate any intermediate abstract representation like the Berkeley tuner and it does not require any predefined strategies for architecture types because it is a high level wavefront pattern tuner that is architecture agnostic, as demonstrated through our tuning results across different kinds of heterogeneous systems. The Berkeley tuner also cannot deal with multiple GPU implementations.

The PATUS framework [18] builds on top of the work done by the Berkeley tuner by providing a high degree of customization and flexibility to the user. It allows the user to choose from predefined strategies regarding how the kernel should be optimized and parallelized. It also lets the user design custom strategies in order to experiment with other algorithms or even discover a better mapping to the underlying hardware. Its modular code generator back-end supports future hardware architectures and programming paradigms by providing communication and synchronization primitives that allow custom definition of hardware-specific characteristics and code generator methods. However, like the Berkeley tuner, it is restricted to single GPU systems.

Autotuning for the stencil pattern has been also been investigated in the multi-GPU framework “PARTANS” [68]. A key difference with our implementation is the absence of dependence between elements in a stencil pattern, which means halo swapping is less frequent for stencils distributed over multiple GPUs than for wavefronts. A three tier multi-GPU+ multi-core + cluster framework stencil tuning framework has been investigated [93] for mesh-based applications. It accepts user supplied code and executes it on multiple GPU by generating CUDA code for GPU and distributed CPU code using MPI and OpenMP. It also provides C++ classes to write GPU-GPU communication effectively as shown in Figure 7.5. Experimental results of carrying out diffusion computation using two NVIDIA Tesla K20X GPUs was found to be 1.4 times faster than hand tuned code. Our framework is restricted to two tier of parallelism - multicore + multi-GPU. However, our GPU kernel code is based on portable OpenCL.



(a) Multi-GPU computing by using both MPI and OpenMP.

(b) Intra-node GPU-GPU communication by OpenMP threads.

(c) Inter-node GPU-GPU communication by MPI

Figure 7.5: A three tier parallel (multicore + multi-GPU + cluster) stencil autotuning framework. The above figure has been adopted from [93]

7.3 Compiler Space Tuners and Optimization using Machine Learning

In this section we discuss some compiler based autotuners and how machine learning is used for compiler space optimization. Compiler based autotuners are based on polyhedral models, loop tiling approaches and time blocking schemes [47]. Polyhedral models are representations of programs that allow compilers to perform complex transformations such as loop nest optimizations and parallelizations in mathematically rigorous ways, ensuring correctness of code. CHiLL [17] is such a polyhedral loop transformation and code generation framework. It provides a convenient high-level scripting interface to the compiler for simplifying code generation and varying optimization parameter values that can then be processed by a search engine, as discussed later in section 7.4. Pluto [12], is an automatic parallelization and locality optimization tuner based on the polyhedral model and is based on minimization of a unified cost function that incorporates aspects of both intra-tile locality and inter-tile communications of the data. The polyhedral model is used for both determining good tile sizes as well for auto-parallelization on CPUs. In contrast to these compiler based tuners, our framework uses machine learning techniques to determine good CPU and GPU tile sizes.

Machine learning techniques have also been utilized to efficiently explore the compiler co-design space by Grewe [27]. The earlier method of compiler optimization involved empirical ad-hoc changing of flags. This is now automated through machine learning and the general idea behind it is to build a learning model that predicts the right optimizations for specific

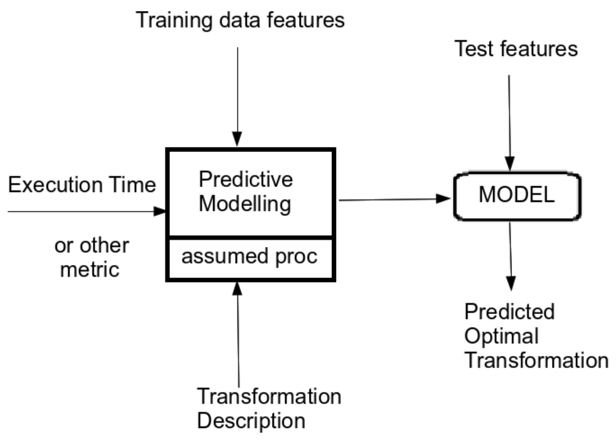


Figure 7.6: Predictive Modeling in Compiler Optimization, adopted from [27]

combinations of programs and platforms. In Grewe's work, decision tree models are used to select either multi-core CPU or GPU implementation and not a hybrid CPU + multi-GPU setup as is done in our framework. His work also focuses on applying machine learning for low level program optimizations (like learning to schedule, loop unroll etc). The trained machine learning models predicted the optimal number of threads for these low level combinations. However, the accuracy of most models in this project were usually in the range of 30-55 percent.

7.4 Dynamic Self Adaptive Tuners

In the final section of this chapter we discuss adaptive tuners.

The Active Harmony framework [55, 19] uses the greedy or Nelder Mead algorithm to search a high dimensional space and the tuning results are then treated as a new experience to be stored in a data characteristics database for future reference. It is an application level auto tuner that uses machine learning in its data analyzer component as illustrated in Figure 7.7.

When input data is fed into the system, the data analyzer examines a small number of sample requests to explore the feature space of the input data using a user supplied method. Once these characteristics are identified, the prior known tuning experience associated with the input request characteristics are retrieved from the data characteristics database. For those input data with characteristics that have never been seen before, the tuning system will try different configurations from scratch on the high dimensional search space (using the greedy or the Nelder Mead algorithm). The tuning results are then treated as a new experience and are used to update the data characteristics database for future reference. The classification mechanism of the Active Harmony tuner is the least square error technique (discussed earlier as part of Linear Regression in subsection 2.4.4), which is used to retrieve

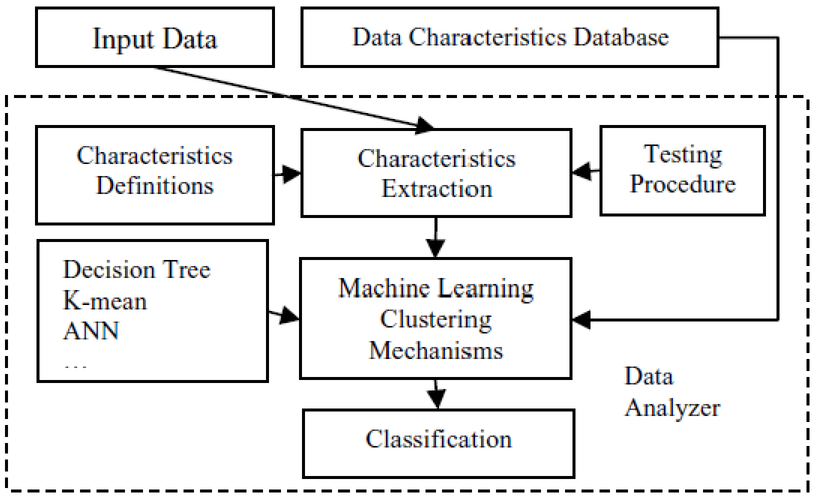

Input Data
Data Characteristics Database
Characteristics Definitions
Characteristics Extraction
Testing Procedure
Decision Tree
K-mean
ANN
...
Machine Learning Clustering Mechanisms
Data Analyzer
Classification

Figure 7.7: The data analyzer component of the Active Harmony dynamic autotuner, adopted from [19]

configurations from previously stored experiences in the data characteristics database. Active Harmony then uses those retrieved configurations to setup the system being tuned.

Dynamic autotuning based on machine learning is also observed in the work of Tiwari et al [98]. Their auto tuner combines Active Harmony's improved parallel search backend with the CHiLL compiler transformation framework to generate in parallel a set of alternative implementations of computation kernels and automatically select the one with the best-performing implementation. Their improved Active Harmony system consists of a parallel search algorithm in place of the original Nelder Mead simplex algorithm for searching the optimization space. This algorithm, called the Parallel Rank Ordering (PRO) Algorithm, deals with high-dimensional search spaces with unknown objective functions. It is essentially a parallel version of a class of direct search algorithms known as the Generating Set Search (GSS) methods that traverse constrained spaces. PRO converges to a solution faster than its sequential counterparts.

Dynamic compiler based autotuning requires a compiler to be able to generate different codes rapidly during the search phase by adjusting parameter values, without time consuming compiler reanalysis. It is also beneficial to have a modular framework so that the compiler can interface to a separate parameter search engine. These ideas led to the deployment of CHiLL, a polyhedral loop transformation and code generation framework (mentioned earlier in section 7.3) whose optimization choices are passed to the search engine. In order to carry out transformation of loops, Active Harmony's search-kernel requests CHiLL's code-generator to create code variants with given sets of parameters. Once compiled, the optimization driver runs the code variants in parallel on the target architecture. The search-kernel bases its simplex transformation decisions on performance metrics. The learning mechanism of storing new experiences and retrieving them through least square error classification mechanism remains unchanged. The performance of this framework generated code is comparable to the fully automated version of the ATLAS library for the tested kernels and is 1.4 to 3.6 times faster than the native Intel compiler without search. Our offline autotuning framework differs from this online tuning framework and we tune high level pattern specific parameters using machine learning techniques to set optimal values for those parameters.

Another adaptive tuner is an adaptive implementation of a task farm known as *Self adaptive farms*. These are based on a single round scheduling algorithm called *Dynamic Deal* [46]. This algorithm is specifically aimed at improving the performance of task farms in computational grids by getting compile time data which is then used at run time to adapt the Task Farm to different load and network conditions in the grid. The task farm dynamically schedules what tasks should be allocated to the workers and then adaptively determines the optimal size of the task for each worker. This kind of self-adaptation reduces the execution time of the task farm application. Determining the task size forms the core of the adaptive strategy. This is based on parameters like CPU availability which affects computation, bandwidth which determines the speed and throughput of farmer-worker communication and latency which affects the time taken to communicate messages between farmer and worker. Unlike our work, here the compile time profile information pertains to low level details like CPU availability and network bandwidth. Also, it is restricted to the computational grid architecture and is not suitable for heterogeneous systems.

7.5 Conclusion

In this chapter we reviewed various types of autotuning frameworks and contrasted their design and performance characteristics with our domain specific tuning of wavefront parameters at the high level of a skeleton. We highlighted autotuners for other high level patterns like task farms and stencils. We discussed advances in multi-GPU + multicore and even multi-node auto tuning framework for stencils. We explored application of machine learning techniques in setting optimal values of tunable parameters in some of these auto-tuning frameworks and concluded the chapter with a discussion of online autotuners .

Chapter 8

Conclusion

Our work has been a case study in designing an efficient machine learning based autotuning framework for applications following a specific parallel pattern. In our study, the pattern is the wavefront and the target architectures are heterogeneous systems comprising multicore CPUs and multiple GPU accelerators.

Our autotuning framework addressed the twin challenges of problem decomposition and distribution by providing a layer of abstraction which partitioned and distributed work across systems, and enabled performance portability by predicting optimal values for tunable parameters. These are very sensitive to different combination of problem instances and heterogeneous systems. Tunable parameters included choosing the number of GPU accelerators, controlling the amount of computation to be offloaded onto GPU accelerators, choosing the number of boundary elements to be swapped between GPU accelerators, and tiling for both CPU and GPU memories. These were evaluated against the best performing points obtained from exhaustively traversing the optimization search space.

The next section in this chapter summarizes our contributions. This is followed by critical analysis of areas of improvement. We conclude with a brief discussion of possible future work.

8.1 Contributions

There are two main contributions of this thesis.

Wavefront Skeletons targeting heterogeneous architectures

Our thesis presents the first definition and implementation of a flexible, heterogeneous abstraction of the wavefront pattern that targets multicore CPUs and multiple GPUs, and addresses two and three dimensional problems. By encapsulating tedious parallel boilerplate code, we allow the application program to focus on only application specific issues in 2D and 3D wavefront patterns. This level of abstraction also allowed us to identify high level parameters that can affect the runtime of applications implemented using our framework across different heterogeneous systems.

Effective machine learning for tuning wavefronts

Our thesis is also the first demonstration that such a scheme can be effectively autotuned using machine learning, to give portability across a wide range of wavefront applications and architectures. As an extension, this also provides a demonstration that pattern information can be usefully exploited in autotuning.

We used regression and support vector machines to tune 2D wavefronts in single GPU + multicore CPU setting in chapter 4. We employed universal and class specific tuners. The class specific tuners had some prior knowledge of the application as they were built separately from training samples of each application and validated against distinct test sets of those application. They predicted with high accuracy, leading to speedup from autotuning being 91% of the optimal speedup found from exhaustive search. However, our universal tuner which was built from training instances from all applications was not so successful, with auto tuned speedup being 64% of the optimal. This led us to experiment with a learning technique that was better suited to our data. An exhaustive analysis of the optimization space revealed a saw-tooth like shape, meaning the best points were arranged as per a piecewise-defined function. The M5 Decision Tree was the learning technique most suited to such a function comprising of multiple sub-functions applied to specific intervals of the main function's domain, as discussed in chapter 5. This tuner was able to obtain 98% of optimal speedup. We also tuned 3D wavefront applications with an impressive 99.5% of the optimal speedup as described in chapter 6. The average speedup from all such tuning across single GPU and multi GPU systems was 87% of the optimal speedup.

8.2 Critical Analysis

In this section we discuss the limitations of our autotuner and lessons learned from our past experiments.

The optimization space was explored manually through regular sampling but we could have used more sophisticated Monte Carlo methods [36] based on repeated random sampling to find the best performing points and generate the shape of our search space. We also learned from the past performance of our naive linear regression tuner and that decision tree based learners were better suited to our data. The same held true in selecting training instances for these models. When dealing with single GPU autotuning, we gathered training instances from all wavefront applications (that included real world applications) to create a "universal tuner" which performed rather poorly. So, for our multiple GPU autotuner and 3D wavefront autotuner, we trained on instances drawn solely from our synthetic application that spanned a wide range of task granularity and problem sizes. This coupled with decision trees learners provided very high accuracy, averaging 97% of optimal performance. Thus we

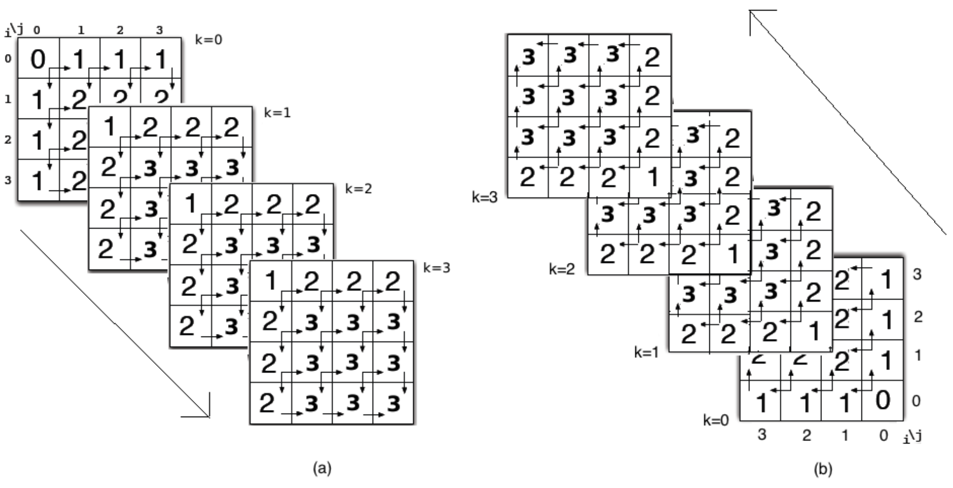


Figure 8.1: (a)Forward Sweep dependency array for 3D Wavefront (b) Reverse Sweep for same.

did not require any real application during the training phase, which is again a strength of our pattern-oriented approach.

In terms of autotuning, we are restricted to offline tuning strategy. While this is effective in predicting optimal values when the training sample from synthetic application is large enough to cover a range of real world 3D wavefront applications, there is always the chance of encountering some application which lies outside our training space. Hence an online tuner built in the same manner as the Active Harmony tuner, would add the newly encountered space or experience into its list of known experiences and then be able to tune a wider range of wavefront applications dynamically.

Another framework limitation was that we only tested applications that had dependency relations in the immediate neighborhood. By immediate neighborhood we refer to elements that are one diagonal away such as *north* and *west* elements, or two diagonals away like the *north-west* elements. Most wavefronts follow this sort of non serial monadic recurrence relation but there could be some that compute on data from elements that are more than two diagonals away.

Our autotuner also does not handle reverse sweeps, which was the reason behind limiting our 3D wavefront application to lower triangulation instead of lower upper triangulation. A reverse sweep would require the dependency to be built in the opposite direction as illustrated in figure Figure 8.1.

8.3 Future work

A possible expansion to our work is to incorporate three tier parallelism comprising of clusters of nodes with each node consisting of multiple GPU accelerators to scale our framework to handle larger problems and provide better speedup from three tier parallelism. This would be similar to the three tier multi-GPU+ multi-core + cluster framework stencil tuning framework of [93] for mesh-based applications.

We would also make our framework more flexible by adding support for sweeps in any direction of the computation grid and provide better customization by allowing access to elements more than two diagonals away. We would also increase the scope of our autotuner to handle more dynamic programming problems, which are not restricted to non serial monadic variety of wavefronts. Our autotuning could also be improved by utilizing sophisticated sampling methods for the training phase and by deploying dynamic online tuning to address the limitations of our offline tuning strategy.

Finally, we could explore alternatives to GPU accelerators for high performance computing with more recent many core developments such as AMD APUs [43] and Intel MIC [42]. We conclude our work with these new developments which are briefly explained in the next subsections.

8.3.1 Accelerated Processing Unit

An accelerated processing unit like the AMD Fusion integrates the CPU and GPU on a single die. This allows the CPU memory management unit (that translates virtual memory addresses to physical addresses) and the equivalent GPU input/output memory management unit to share the same address space. The advantage from such a design is the time saved from not requiring transfer of data from GPU to host memory and back through PCI-E. This design also allows higher performance since the GPU can access CPU cache data and the memory is fully cache coherent between CPU and GPU. Other performance improving features include a preemptive scheduler that can interrupt long running tasks for low latency access to GPU, and context switching ability of the GPU.

8.3.2 Many Integrated Core Architecture

Intel's Xeon Phi coprocessor (a processor that supplements the main CPU by offloading compute intensive tasks from the CPU) is an example of Intel's many integrated core architecture (MIC). Each co-processor consists of 61 cores and four hardware threads per core (244 threads)[42]. To reduce energy consumption, the clock speed is kept low at 1.01 GHz, but high performance is achieved through the aggregate throughput of 244 threads. Apart from task level parallelism obtained from 4 threads per core, the availability of wide vector units (512 bit) for SIMD computations enables users to develop vector intensive code (such as vectorized loops) to exploit high levels of data parallelism. MIC lets the user choose from native, symmetric and offload execution. Under native execution there is only one executable that runs on the MIC architecture which is useful for benchmarking and analysis. In symmetric mode the code is compiled twice, one for host and once for MIC. Thus, different work items can be assigned to CPUs and MICs with host to MIC communication (and host-host, MIC-MIC) being carried out in MPI. The offload execution mode is similar to shared memory directive based models where the directives indicate the data and methods that are offloaded from CPU to MIC for execution, and the code needs to be compiled only once.

Bibliography

[1] A. M. Aji and W. Feng. *Accelerating Data-Serial Applications on Data-Parallel GPGPUs: A Systems Approach.* Technical Report TR-08-24, Computer Science, Virginia Tech, 2008.

[2] G. S. Almasi. Overview of parallel processing. *PARALLEL-COMPUTING*, 2(3):191–203, nov 1985.

[3] C. Alves, E. Caceres, F. Dehne, and S. Song. A parallel wavefront algorithm for efficient biological sequence comparison. 2668:973–973, 2003.

[4] John Anvik, Steve Macdonald, Duane Szafron, Jonathan Schaeffer, Steven Bromling, and Kai Tan. Generating parallel programs from the wavefront design pattern. In *Proceedings of the 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments, April 2002. On CD*, pages 1–8. Society Press, 2002.

[5] John Anvik, Steve Macdonald, Duane Szafron, Jonathan Schaeffer, Steven Bromling, and Kai Tan. Generating parallel programs from the wavefront design pattern. In *Proceedings of the 7th International Workshop on High-Level Parallel Programming Models and Supportive Environments, April 2002. On CD*, pages 1–8. Society Press, 2002.

[6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[7] S. Orlando S. Pelagatti M. Vanneschi B. Bacci, M. Danelutto. P3l: a structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[8] Eddie and the ecdf. `https://www.wiki.ed.ac.uk/display/ecdfwiki/Eddie+and+the+ECDF`.

[9] Tianhe-1a: National supercomputing center in tianjin. `http://www.top500.org/featured/systems/tianhe-1a-national-supercomputing-center-in-tianjin/`.

[10] Lattice gauge theory software research and development. `http://usqcd.fnal.gov/software.html`.

[11] Blaise Barney. Posix threads programming. `https://computing.llnl.gov/tutorials/pthreads/`.

[12] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234. ACM, 2008.

[13] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[14] Murilo Boratto, Pedro Alonso, Domingo Giménez, Marcos Barreto, and Karolyne Oliveira. Auto-tuning methodology to represent landform attributes on multicore and multi-GPU systems. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 125–132, New York, NY, USA, 2013. ACM.

[15] Mpi: A message-passing interface standard version 2.2. `http://www.mpi-forum.org/`.

[16] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 13:1–13:11, New York, NY, USA, 2011. ACM.

[17] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. *U. of Southern California, Tech. Rep*, pages 08–897, 2008.

[18] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[19] I-Hsin Chung and Jeffrey K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 30–, Washington, DC, USA, 2004. IEEE Computer Society.

[20] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA, USA, 1991.

[21] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.

[22] G. Contreras and M. Martonosi. Characterizing and improving the performance of Intel Threading Building Blocks. *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57–66, 2008.

[23] Tim Cook. Comparing the ultrasparc t2 plus to other recent sparc processors. `https://blogs.oracle.com/timc/entry/comparing_the_ultrasparc_t2_plus`.

[24] The openmp api specification for parallel programming. `http://openmp.org/wp/`.

[25] John Darlington, Yi-ke Guo, Hing To, and Jin Yang. Functional skeletons for parallel coordination. *EURO-PAR '95 Parallel Processing*, 966:55–66, 1995.

[26] Heikki Mannila David J. Hand and Padhraic Smyth. *Principles of Data Mining.* MIT Press, 2001.

[27] D.Grewe. Adaptively mapping parallelism based on system workload using machine learning. 2009.

[28] A.J. Dios, R. Asenjo, A. Navarro, F. Corbera, and E.L. Zapata. Evaluation of the task programming model in the parallelization of wavefront problems. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 257–264, Sept 2010.

[29] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[30] R.G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, Feb 2010.

[31] Intel Threading Building Blocks (Intel TBB) 3.0 for opensource. `http://www.threadingbuildingblocks.org/`.

[32] Introductory applied machine learning course homepage. `http://www.inf.ed.ac.uk/teaching/courses/iaml/`.

[33] Joel Falcou and Jocelyn Serot. Quaff: efficient c++ design for parallel skeletons. *Parallel Computing*, 32(7-8):604–615, 2006.

[34] Jianbin Fang, A.L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225, Sept 2011.

[35] John Feehrer, Paul Rotker, Milton Shih, Paul Gingras, Peter Yakutis, Stephen Phillips, and John Heath. Coherency hub design for multisocket sun servers with coolthreads technology. *IEEE Micro*, 29:36–47, 2009.

[36] George S Fishman. *Monte Carlo*. Springer, 1996.

[37] Eibe Frank and Ian H. Witten. *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*. Morgan Kaufmann, 2005.

[38] Diana Franklin and Frederic T. Chong. Challenges in reliable quantum computing. In Sandeep K. Shukla and R.Iris Bahar, editors, *Nano, Quantum and Molecular Computing*, pages 247–266. Springer US, 2004.

[39] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[40] Cray cs-storm. `http://www.cray.com/sites/default/files/resources/CrayCS-Storm.pdf`.

[41] Cray xe6. `http://www.cray.com/sites/default/files/resources/CrayXE6Brochure.pdf`.

[42] Xeon phi many integrated core. `https://www.microway.com/download/datasheet/Intel_Xeon_Phi_Coprocessor_Family_Brief.pdf`.

[43] Apu 101: All about amd fusion accelerated processing units. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/apu101.pdf`.

[44] Intel sdk for opencl applications 2012. `https://software.intel.com/sites/landingpage/opencl/optimization-guide/Basic_Concepts.htm`.

[45] Isaac Gelado, John E Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 347–358. ACM, 2010.

[46] Horacio González-Vélez. Self-adaptive skeletal task farm for computational grids. *Parallel Computing*, 32(7–8):479–490, 2006.

[47] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *Proceedings of 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 106–111. IEEE, 1998.

[48] D. Grosu. Dynamic programming (dp). `http://www.ece.eng.wayne.edu/~sjiang/ECE7610-winter-11/lecture-6.pdf`.

[49] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. On clustering validation techniques. *Journal of Intelligent Information Systems*, 17(2-3):107–145, 2001.

[50] S. D. Hammond, G. R. Mudalige, J. A. Smith, and S. A. Jarvis. Performance prediction and procurement in practice: Assessing the suitability of commodity cluster components for wavefront codes. *IET SOFTW.*, 3(6):509–521, 2009.

[51] Rolf Hempel and David W. Walker. The emergence of the mpi message passing standard for parallel computing. *Comput. Stand. Interfaces*, 21(1):51–62, 1999.

[52] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman Publishers, 2007.

[53] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, 2008.

[54] H.Kuchen and M.Poldner. On implementing the farm skeleton. *Parallel Processing Letters*, 18(1):117–131, 2008.

[55] Jeffrey K. Hollingsworth and Peter J. Keleher. Prediction and adaptation in active harmony. *Cluster Computing*, 2:195–205, July 1999.

[56] CSA Dept IISc. All pairs shortest paths problem: Floyd's algorithm. `http://lcm.csa.iisc.ernet.in/dsa/node164.html`.

[57] Haoqiang Jin, Michael Frumkin, and Jerry Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[58] RC Whaley JJ Dongarra, A Petitet. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.

[59] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[60] Shoaib Kamil, Cy Chan, Samuel Williams, Leonid Oliker, John Shalf, Mark Howison, and E. Wes Bethel. A generalized framework for auto-tuning stencil computations. In *In Proceedings of the Cray User Group Conference*, 2009.

[61] Y Kreinin. SIMD< SIMT< SMT: parallelism in nvidia GPUs. `http://www.yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html`, 2011.

[62] H. Kuchen. The skeleton library web pages. `http://danae.uni-muenster.de/lehre/kuchen/Skeletons/`.

[63] Herbert Kuchen. A skeleton library. *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, 2002.

[64] Nasser Kurd, Jonathan Douglas, Praveen Mosalikanti, and Rajesh Kumar. Next generation intel® micro-architecture (nehalem) clocking architecture. In *VLSI Circuits, 2008 IEEE Symposium on*, pages 62–63. IEEE, 2008.

[65] John Levesque, Jeff Larkin, Martyn Foster, Joe Glenski, Garry Geissler, Stephen Whalen, and et al. Understanding and mitigating multicore performance issues on the amd opteron architecture. 2007.

[66] Jonathan Levin. Solution concepts. `http://web.stanford.edu/~jdlevin/Econ%20286/Solution%20Concepts.pdf`, 2006.

[67] Xiaoming Li, María Jesús Garzarán, and David Padua. Optimizing sorting with machine learning algorithms. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6. IEEE, 2007.

[68] Thibaut Lutz, Christian Fensch, and Murray Cole. Partans: An autotuning framework for stencil computation on multi-GPU systems. *ACM Trans. Archit. Code Optim.*, 9(4):59:1–59:24, January 2013.

[69] Intel Threading Building Blocks. `http://threadingbuildingblocks.org/`.

[70] T. Mattson. The future of many core computing: a tale of two processors. `http://cseweb.ucsd.edu/classes/fa12/cse291-c/talks/SCC-80-core-cern.pdf`, 2010.

[71] Oliver A. McBryan. An overview of message passing environments. *Parallel Computing*, 20:417–444, 1994.

[72] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann, 2012.

[73] M.Cole. eskel: The edinburgh skeleton library api reference manual. `http://homepages.inf.ed.ac.uk/mic/eSkel`.

[74] Tom Mitchell. *Machine Learning.* McGraw Hill, 1997.

[75] Siddharth Mohanty. Task farm optimization through machine learning. Master's thesis, Institute of Computing Systems Architecture, Informatics Department, University of Edinburgh, 2010.

[76] Siddharth Mohanty and Murray Cole. Autotuning wavefront abstractions for heterogeneous architectures. In *Applications for Multi-Core Architectures (WAMCA), 2012 Third Workshop on*, pages 42–47. IEEE, 2012.

[77] Siddharth Mohanty and Murray Cole. Autotuning wavefront applications for multicore multi-GPU hybrid architectures. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'14, pages 1:1–1:9, New York, NY, USA, 2014. ACM.

[78] Andrew W Moore. Cross-validation for detecting and preventing overfitting. *School of Computer Science Carneigie Mellon University*, 2001.

[79] Gihan R Mudalige, Mary K Vernon, and Stephen A Jarvis. A plug-and-play model for evaluating wavefront computations on parallel architectures. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–14. IEEE, 2008.

[80] Single chip cloud computing platform overview. `http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-platform-overview-paper.html`.

[81] Elona Pelez. Parallelism and the crisis of von neumann computing. *Technology in Society*, 12(1):65 – 77, 1990.

[82] Simon J Pennycook, Simon D Hammond, Gihan R Mudalige, Steven A Wright, and Stephen A Jarvis. On the acceleration of wavefront applications using distributed many-core architectures. *The Computer Journal*, 55(2):138–153, 2012.

[83] Simon J Pennycook, Simon D Hammond, Steven A Wright, JA Herdman, Ian Miller, and Stephen A Jarvis. An investigation of the performance portability of opencl. *Journal of Parallel and Distributed Computing*, 73(11):1439–1450, 2013.

[84] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[85] Moore's law: how long will it last? `http://www.techradar.com/news/computing/moore-s-law-how-long-will-it-last--1226772`.

[86] Excerpts from a conversation with gordon moore: Moore's law. `http://large.stanford.edu/courses/2012/ph250/lee1/docs/Excepts_A_Conversation_with_Gordon_Moore.pdf`.

[87] J Ross Quinlan. Combining instance-based and model-based learning. In *ICML*, pages 236–243, 1993.

[88] M. Riedmiller. Machine learning generalisation in multilayer perceptron. `http://ml.informatik.uni-freiburg.de/_media/teaching/ss10/06_mlp_generalisation.printer.pdf`.

[89] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3d scientific computations. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[90] Philip E. Ross. 5 commandments. `http://spectrum.ieee.org/semiconductors/materials/5-commandments/2`.

[91] Aamir Shafi, Bryan Carpenter, and Mark Baker. Nested parallelism for multi-core hpc systems using java. *J. Parallel Distrib. Comput.*, 69(6):532–545, 2009.

[92] Hongzhang Shan, Jaswinder P. Singh, Leonid Oliker, and Rupak Biswas. Message passing and shared address space parallelism on an smp cluster. *Parallel Computing*, 29(2):167 – 186, 2003.

[93] Takashi Shimokawabe, Takayuki Aoki, and Naoyuki Onodera. A high-productivity framework for multi-GPU computation of mesh-based applications. *HiStencils 2014*, page 23, 2014.

[94] Lorna Smith and Mark Bull. Development of mixed mode mpi/openmp applications. *Sci. Program.*, 9(2,3):83–98, 2001.

[95] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[96] Thomas Sterling. An overview of cluster computing. pages 15–29, 2002.

[97] Julien C. Thibault and Inanc Senocak. Accelerating incompressible flow computations with a pthreads-cuda implementation on small-footprint multi-GPU platforms. *The Journal of Supercomputing*, 59(2):693–719, 2012.

[98] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[99] Paul E. Utgoff. *Machine learning of inductive bias.* Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 1986.

[100] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[101] Eric W. Weisstein. Lu decomposition. `http://mathworld.wolfram.com/LUDecomposition.html`.

[102] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers.* PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[103] Lexing Xie, Shih-Fu Chang, A. Divakaran, and Huifang Sun. Unsupervised discovery of multilevel statistical video structures using hierarchical hidden markov models. In *Multimedia and Expo, 2003. ICME '03. Proceedings. 2003 International Conference on*, volume 3, pages 29–32, July 2003.

[104] Li Yu, Christopher Moretti, Andrew Thrasher, Scott Emrich, Kenneth Judd, and Douglas Thain. Harnessing parallelism in multicore clusters with the all-pairs, wavefront, and makeflow abstractions. *Cluster Computing*, 13:243–256, September 2010.