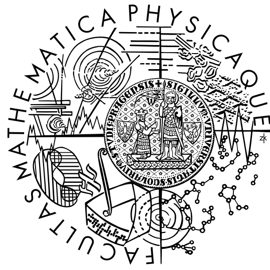

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

David Ligr

Parallel Evaluation of Numerical Models for Algorithmic Trading

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Martin Kruliš, Ph.D.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

Title: Parallel Evaluation of Numerical Models for Algorithmic Trading

Author: David Ligr

Department: Department of Software Engineering

Supervisor: RNDr. Martin Kruliš, Ph.D., Department of Software Engineering

Abstract: This thesis will address the problem of the parallel evaluation of algorithmic trading models based on multiple kernel support vector regression. Various approaches to parallelization of the evaluation of these models will be proposed and their suitability for highly parallel architectures, namely the Intel Xeon Phi coprocessor, will be analysed considering specifics of this coprocessor and also specifics of its programming. Based on this analysis a prototype will be implemented, and its performance will be compared to a serial and multi-core baseline pursuant to executed experiments.

Keywords: parallelization, GPU, Xeon Phi, algorithmic trading, support vector machines

First of all, I would like to express my gratitude to my supervisor RNDr. Martin Kruliš, PhD. for his comments, remarks and support. Further, I am very grateful to all my friends, who read this work and helped me with corrections.
Last but not least, I would like to thank my family for supporting me during the studies, especially my wife Eva for her patience and encouragement.

Contents

1 Introduction **4**
1.1 Performance and Parallelism 5
1.2 Objectives 6

2 OpenCL **7**
2.1 Structure of OpenCL 7
2.1.1 Language Specification 7
2.1.2 Platform Layer and Runtime API 8
2.2 Platform Model 8
2.3 Execution Model 9
2.3.1 Kernel Execution on a Compute Device 9
2.3.2 Host Program 11
2.4 Memory Model 15
2.4.1 Accessing Shared Memory Simultaneously 17
2.5 Best Practices 18
2.5.1 Work Decomposition 18
2.5.2 Synchronization 18
2.5.3 Reusing of Data 18
2.5.4 Data Layout 18

3 Intel Xeon Phi **20**
3.1 Architecture 20
3.1.1 Vector Processing Units 21
3.1.2 Memory Architecture 22
3.2 OpenCL Programming 24
3.2.1 Kernel Execution on Intel Xeon Phi Coprocessor 24
3.2.2 Local Memory 26
3.2.3 Comparison with GPUs 26

4 AT Model **28**
4.1 Technical Analysis 28
4.1.1 Technical Indicators 29
4.1.2 Univariate Analysis 29
4.1.3 Multivariate Analysis 30
4.1.4 Supervised Learning Algorithms 30
4.2 Support Vector Machine 32
4.2.1 Theory of SVMs 33

4.2.2 Kernel Function . . . 35
4.2.3 SVM in Regression . . . 37
4.3 Existing AT Model . . . 37

5 Implementation **40**
5.1 Analysis . . . 40
5.1.1 Preprocessing . . . 41
5.1.2 Approaches to Parallelization . . . 41
5.2 Architecture Specific Parallelization . . . 44
5.2.1 Multi-Core Parallelization . . . 44
5.3 Many-Core Parallelization . . . 45
5.3.1 Evaluating Kernel Method In Parallel . . . 46
5.3.2 Evaluating Multiple Blocks Simultaneously . . . 48
5.3.3 Parallel Aggregation . . . 49
5.3.4 Preprocessing . . . 51
5.3.5 Data Layout . . . 51
5.3.6 Command-Queues and Kernel Synchronization . . . 52
5.4 Programming Language, Data Formats, and Libraries . . . 52
5.4.1 Data Types . . . 53
5.4.2 Parallel Programming in .NET . . . 53
5.4.3 OpenCL Host Bindings in .NET . . . 54
5.5 Implementation for GPUs . . . 55

6 Experimental Results **57**
6.1 Experimental Methodology . . . 57
6.1.1 Execution Time . . . 58
6.1.2 Correctness of Measured Times . . . 58
6.1.3 Hardware Specification . . . 59
6.1.4 Test Data . . . 60
6.1.5 Testing Configuration . . . 60
6.2 Performance . . . 60
6.2.1 Measured Times . . . 61
6.2.2 Scalability . . . 63
6.3 Cost of Modifications . . . 65

7 Conclusion **67**
7.1 Future Work . . . 67

Bibliography **69**

List of Figures **72**

List of Tables 73

List of Abbreviations 74

Attachments 75

1. Introduction

Trading on stock markets mainly consists of selling and buying trading instruments, such as shares, obligations, or derivatives. All these operations are performed pursuant to many aspects, which can be grouped into two disjoint sets: rational aspects and irrational aspects. The examples of rational aspects include a progression of a price of the given trading instrument, progressions of prices of trading instruments related to the given trading instrument, or a due date of the corresponding obligation. The irrational aspects are for example intuition and feelings of the trader.

Up to the 1950s, all activities concerning buying and selling instruments were carried out by traders. Consequently, decisions could have been influenced by irrational aspects as well as rational aspects. Thanks to that, an unexpected drop of price of owned trading instrument might have led to a lossy decision, which was based on fear and nervosity of the trader rather than indicias coming from an analysis of rational aspects. This was one of the most important reasons why Harry Markowitz came in his dissertation thesis [1] with the idea of applying mathematical concepts to stock markets. This idea combined with computers development led to emergence of algorithmic trading (AT). From that time on, trading operations can be assisted by algorithmic trading or fully executed by it.

In algorithmic trading, mathematical models are transformed into computer algorithms. This allows processing of enormous amount of data in a short period of time and therefore it is possible to quickly react to occurring situations. This is one of the most important reasons why algorithmic trading was utilized in up to 63% of all trades done on the US stock markets in 2011 (this number was published in The Economist [2]). However, it is important to note that even though a trader leaves part of their activities to a computer program, certain decisions can still be made by the trader themselves. These activities typically include an execution of trading instructions (algorithmic execution), while the trader still makes a decision about their issuing.

High-frequency trading (HFT) is the primary form of algorithmic trading on financial markets. High-frequency trading represents so-called algorithmic decision-making. In algorithmic decision-making, a computer is also responsible for issuing trade requests in contrast to algorithmic execution, in which a trader is responsible for this issuing. The main characteristics of high-frequency trading include a fast reaction to events occurring on the market and also hundreds to thousands of orders and messages issued to the market per seconds.

1.1 Performance and Parallelism

Companies dealing with algorithmic trading utilize so-called co-location, i.e. they place their servers right into the buildings where the stock markets are situated. A close physical proximity leads to shorter communication paths and to lower communication latency.

Another technique being used to improve performance is utilization of more powerful hardware. Until the beginning of the 21st century, the power of contemporary processors could have been easily improved by the frequency increase. In that time, the frequency of processors reached a limit of about 3 GHz, beyond which processors consume too much energy and consequently produce too much heat. In order to solve this problem, processor developers had to adopt a new approach to improve the performance. Therefore they shifted their focus on optimizing hardware architecture and on creating multi-core and many-core architectures.

Even though general-purpose processors include a wide range of functional units to respond to any computational demand, only some of these units are utilized in common computation. The problem concerning idle units is that they are still powered, even though far less than utilized ones. Besides general-purpose processors, there are specialized processors, such as graphic cards or parallel accelerators, which are optimized for processing of large blocks of data in parallel. Thanks to this, these processors do not contain such a wide range of functional units, and therefore have better performance per watt than general-purpose processors.

Graphic cards and parallel processors are representatives of so-called many-core architectures. Such architectures allow executing 10-1000 of threads in parallel, and eventually deliver an enormous amount of raw performance. In order to utilize this performance, we have to adopt new ways of designing algorithms, developing applications, and thinking about problems. Applications that utilize that performance are rated both by criteria applicable to single-threaded applications (such as their efficiency or the time complexity) and by scalability with more threads or cores.

Although parallelism can improve throughput in orders of magnitude, it introduces many new issues (such as thread management or synchronization) that do not concern us in single-threaded applications. All these problems have to be addressed properly, in order to create flawless and efficient applications.

In contrast to the fact that parallelism can improve overall computation throughput, it might deteriorate computation latency. Thanks to this, parallelism is not suitable for simple computations, as in these computations latency

interests us more than overall throughput.

1.2 Objectives

In this work we will analyze an existing model for high-frequency trading, which predicts future prices of a trading instrument on the basis of multiple indicators, from the perspective of parallel programming. Based on this analysis, we will implement a prototype, which will efficiently utilize a highly parallel architecture, namely Intel Many Integrated Core (MIC) Architecture, and compare the performance of this prototype implementation to a baseline serial and multi-core CPU algorithm.

In Chapter 2, we will describe OpenCL, which allows us to implement our prototype for the Intel MIC Architecture. Chapter 3 will explain specifics of programming for Intel Xeon Phi, which is the first product based on that architecture. The description of the existing algorithmic trading model and its analysis will follow in Chapter 4. Chapter 5 will be dedicated to the explanation of implementation techniques and details. The empirical results of our research and their interpretation will be presented in Chapter 6.

2. OpenCL

OpenCL (Open Computing Language) [4] is open, royalty-free standard for cross-platform, parallel programming of diverse processors found in personal computers, servers etc. A primary benefit of OpenCL is substantial improvement of speed and responsiveness of a wide spectrum of applications from various market categories, e.g., gaming, scientific and medical software, etc. A second benefit is cross-vendor software compatibility.

When we wish to utilize OpenCL we have to perform even several tasks permitting this utilization. For now, we will describe these tasks at the highest possible level of abstraction. A more elaborate description of the terms stated in the rest of this paragraph is the subject of next subchapters that briefly introduce models constituting OpenCL. The first step to using OpenCL is querying the *platform*, i.e. vendor specific OpenCL implementation, from the *host program* for available *devices* and selecting a subset of them to be utilized for a computation. After that, a *context* containing selected devices is created. The created context is after here utilized for creating a *command-queue*, through which the host program orchestrates a single device within the context, a *kernel*, i.e. a function to be executed on the devices contained in the context, and *memory objects*, which hold input or output data of a kernel invocation. When the command-queue, the kernel, and memory objects are created, the host program submits *commands* to transfer memory objects onto the device, execute the kernel, and transfer yielded data back onto the host to the command-queue.

2.1 Structure of OpenCL

The OpenCL development framework comprises 2 parts:

- Language specification
- Platform layer and runtime API

2.1.1 Language Specification

The language specification describes the syntax and programming interface for writing kernels, i.e. functions to be executed on supported devices in parallel.

```
__kernel void add_matrices(
  constant float* const matrix_a,
  constant float* const matrix_b,
  global float* const matrix_c)
  {
    int r = get_global_id(0);
    int c = get_global_id(1);
    int n = get_global_size(0);
    int index = n*r +c;
    matrix_c[index] = matrix_a[index] + matrix_b[index];
  }
```

Listing 2.1: Source code of kernel performing addition of matrices.

The kernels are written in the OpenCL C [5] programming language. OpenCL C is based on the ISO C99 specification with added extensions and restrictions. Additions include vector data types, vector operations, address space qualifiers and a *kernel* function qualifier, which denotes a kernel function. The source code of a simple kernel function, which encompasses the kernel qualifier and the address space qualifiers, is depicted in Listing 2.1. Restrictions contain the absence of support for function pointers, bit fields, recursion, and variable-length arrays.

2.1.2 Platform Layer and Runtime API

The platform layer API permits the programmer query the system for the existence of OpenCL supported devices. Using this API, a programmer could also control which subset of available devices will constitute the context in any OpenCL application.

The OpenCL runtime API provides the functions to manage objects such as command-queues, memory objects and kernel objects, as well as functionality for executing kernels on one or more devices defined in the context.

2.2 Platform Model

The OpenCL platform model defines a high-level abstraction of any heterogeneous platform used with OpenCL. An OpenCL platform always incorporates a single host which has a general-purpose CPU and one or more OpenCL devices. A schema of such a platform is depicted in Figure 2.1[1].

An OpenCL device can be a CPU, a GPU, or any other accelerator device that

[1]Figures presented in this chapter are based on figures contained in *OpenCL Programming Guide* [6].

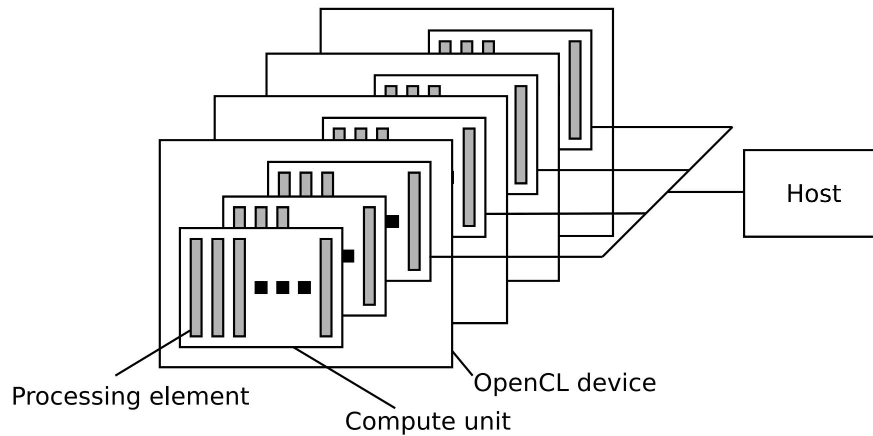

Figure 2.1: A schema of the OpenCL platform model.

is supported by OpenCL. The device is where kernels are executed. Therefore, an OpenCL device is often referred to as a compute device.

An OpenCL device consists of at least one compute unit, which itself is composed of one or more processing elements. These elements execute computational work that is to be executed on a relevant compute device. For instance, in case a compute device is a multi-core CPU, then each of its cores corresponds to a compute unit and slots in SIMD registers correspond roughly to processing elements.

2.3 Execution Model

An OpenCL application consists of two main execution units: host program and kernels. As we mentioned earlier, kernels are basic unit of executable code that runs on one or more OpenCL devices. The host program executes on the host system and uses the runtime API to interact with objects defined within OpenCL.

2.3.1 Kernel Execution on a Compute Device

Before we will describe, how kernels execute on compute devices, we have to introduce data parallelism inasmuch as OpenCL utilizes this kind of parallelism when executing these kernels. Data parallelism refers to scenarios in which the same function executes on different elements of a some collection in parallel. Note that this programming model requires elements in this collection to be independent of each other.

Since data parallelism is about executing a given function on multiple elements of some collection a work associated with execution of that function on a single element conforms to the base unit of work. In OpenCL, this unit of work is

called a *work-item*. The total number of work-items, which execute on processing elements, to be executed is defined by the size of an index space, which is called a *global index space*. This index space is defined when a kernel is queued for execution by the host program.

Work-items are aggregated into work-groups. All work-items in the same work-group are executed together on the processing elements of a single compute device. The reason for execution on a single compute unit is to allow work-items to share resources and synchronize their execution. It is particularly important to realize that the work-items in different work-groups cannot be synchronized, and therefore, accessing the same data by work-items belonging to different work-groups may lead to a malfunction.

Index spaces

As stated, the size of the global index space defines the total number of work-items that require execution. In other words, a work-item is executed for each point in the global index space and, therefore, coordinates of a point uniquely identifies both this point and the associated work-item. These coordinates are commonly referred to as a *global ID* of a work-item.

Since a work-group encompasses related work-items, it provides a more coarse-grained decomposition of the global index space. OpenCL requires all the work-groups to have the same size, i.e. the number of contained elements that divides the global size without a residue. Besides this, a work-group is, just like each work-item, assigned a unique group ID.

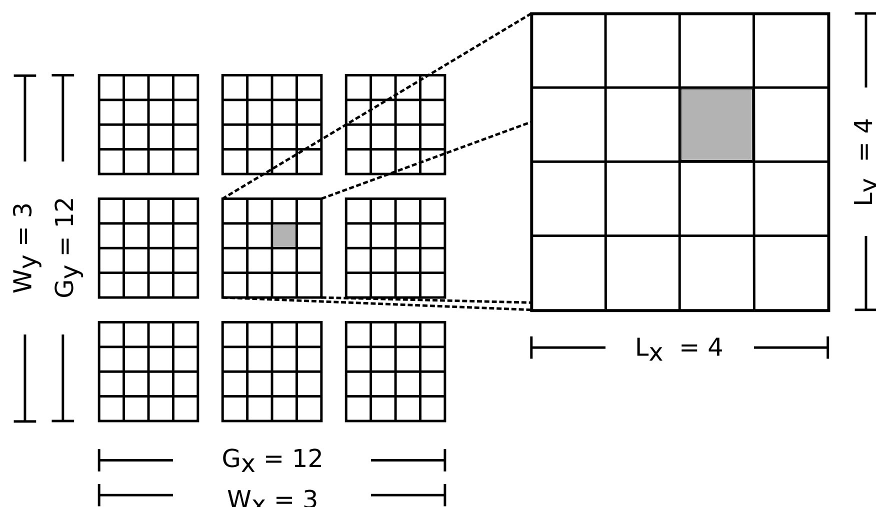

Figure 2.2: An example of how the global IDs, the local IDs, and work-group IDs are related. The shaded block has the global ID of (g_x,g_y) = (6,5) and the local ID of (l_x,l_y) = (2,1).

Besides the global ID, a work-item is assigned a local ID, which is unique

within the work-group containing this work-item. Hence, a work-item can be uniquely identified by its global ID or by a combination of its local ID and its work-group ID. The relation between the local ID and the global ID of a work-item is illustrated in Figure 2.2.

In fact, OpenCL supports up to 3-dimensional index spaces. The supported dimensions correspond to dimensions of entities, e.g., vectors, images, 3D models, whose processing provides opportunities to utilizing data parallelism.

When work-items of a multidimensional index space should be executed by processing elements a mapping of them to these processing elements, i.e. a mapping from a multidimensional space into a one-dimensional space has to be utilized. OpenCL uses the function $f(x, y, z) = x + y \cdot size_x + z \cdot size_x \cdot size_y$, where x corresponds to index in the first dimension and $size_x$ corresponds to the size of the original index space in the first dimension and so on, for this mapping.

2.3.2 Host Program

The host program is responsible for setting up and managing the execution of kernels on OpenCL devices.

In order to dispatch a kernel for execution, the host program has to do following:

1. select an OpenCL platform encompassing devices to be employed in the computation

2. initialize a context using a subset of devices from the selected platform

3. initialize command-queues, program objects, kernels, and memory objects associated with the context

Command-queues, program objects, kernels, and memory objects are essential for any OpenCL application, and therefore, they will be described thoroughly in the rest of this chapter.

Program Object and Kernel

An OpenCL program comprises kernels, data constants and other functions that are non-invokable by the host. A program object encapsulates the program source code or a binary file containing the program executable along with the list of devices for which the executable is built.

When a program object is successfully created, it may be built for one or more devices that are encapsulated by the program object. The reason, why the program objects are built only at runtime, is based on the fact that an OpenCL

programmer writes an application for an end user and does not know on which CPUs, GPUs, or other parallel devices the end user may run the application on. In the other words, a programmer only knows that the target devices conform the OpenCL specification. Therefore, only after defining devices the program is to be executed on, the program object can be appropriately compiled for these devices. In addition, the knowledge of target devices enables the OpenCL compiler to optimize the code in a way suitable for these devices.

```
1 var platform = ComputePlatform.Platforms[0];
2 var context = new ComputeContext(
3   ComputeDeviceTypes.Gpu,
4   new ComputeContextPropertyList(platform),
5   null,
6   IntPtr.Zero);
7 var program = new ComputeProgram(context, clSourceCode);
8 program.Build(null, null, null, IntPtr.Zero);
9 var kernel = program.CreateKernel("addKernel");
```

Listing 2.2: Initialization of a general OpenCL application.

A kernel object, i.e. an object encapsulating a kernel function, is created after the executable has been successfully built in a program object.

All steps required for a creation of a kernel object are stated in Listing 2.2, which contains a snippet of the host source code.

Command-Queues

OpenCL does not support direct communication of the host with devices, instead of this the host communicates with devices by submitting commands to command-queues. A command-queue is created by the host and attached to exactly one OpenCL device.

The order commands submitted to a single command-queue execute is determined by the type of the command-queue. There are 2 types of command-queues:

- **In-order command-queue**: Commands are launched and completed in the same order in which they placed onto the command-queue. In other words, this type guarantees that a command begins only after the preceding commands are finished.

- **Out-of-order command-queue**: Commands are launched in the order that is based on synchronization constraints placed on these commands.

As we have explained above, commands in an in-order command-queue can begin only after all previously enqueued commands are completed. This can decrease performance because of underutilized compute units. Utilization of com-

pute units can be significantly improved by using an out-of-order command-queue, since a command can be launched regardless of whether previously enqueued commands are completed or not. So when a compute unit finishes its work, it can immediately fetch a new command and start its execution. This is called automatic load balancing and it is a well know technique used in the design of parallel algorithms driven by command queues [3].

Regrettably, support of the out-of-order mode is optional, and therefore not implemented on certain platforms. On these platforms, multiple command-queues can be employed to utilize compute units more efficiently. Commands in different command-queues may run concurrently, and only the host can synchronize execution of these commands.

OpenCL supplies three types of commands:

- **Kernel execution**: a command that invokes a kernel for execution on a device, it also specifies the global index space over which the kernel is to be executed
- **Memory transfer**: a set of commands that transfer data between the host and the memory objects, copy the memory objects, or manage memory mappings
- **Synchronization**: a set of commands that constrain order of command execution or synchronize the host execution with a device execution

Memory transfer commands and synchronization commands may be blocking and non-blocking. The OpenCL function call for a non-blocking command returns immediately after the command is enqueued regardless of whether the command is completed or not. On the other hand, the OpenCL function call for a blocking counterpart returns once the command is completed.

The kernel execution commands and the memory transfer commands are used in each OpenCL application. A basic usage of these commands is depicted in Listing 2.3. On the lines 1-2, a command-queue associated with the first device within the context is initialized. The lines 3-5 contain initialization of memory objects. These memory objects are set to the arguments of the kernel, on the following 3 lines. The lines 9-10 include enqueuing of writing of input data, which are obtained elsewhere, to the input memory objects. This implies that buffers *bufferA* and *bufferB* represent input memory objects while the last buffer represents an output memory object. On the line 11, the kernel is submitted to the command-queue for execution. On the last line, reading of data yielded by the kernel execution is enqueued.

```
var commandQueue = new ComputeCommandQueue(context,
  context.Devices[0], ...);
var bufferA = new ComputeBuffer<float>(...);
var bufferB = new ComputeBuffer<float>(...);
var bufferC = new ComputeBuffer<float>(...);
kernel.SetMemoryArgument(0, bufferA);
kernel.SetMemoryArgument(1, bufferB);
kernel.SetMemoryArgument(2, bufferC);
commandQueue.WriteToBuffer(matrixA, bufferA, ...);
commandQueue.WriteToBuffer(matrixB, bufferB, ...);
commandQueue.Execute(kernel, ...);
commandQueue.ReadFromBuffer(bufferC, ref matrixC, ...);
```

Listing 2.3: Source code of an application submitting a kernel for execution.

Synchronization commands are used to constrain the order of execution of multiple commands or of the host and a device. Emphasize that the order of execution of commands issued to the same in-order command-queue is fixed by the order in which commands were enqueued, and thus, it is meaningless to constrain it further using synchronization commands.

The first command that can be used to synchronize execution of multiple commands is *clEnqueueBarrier*. This command enqueues a synchronization point ensuring that all commands enqueued to the same command-queue before have finished execution before following commands begin execution.

As stated, using *clEnqueueBarrier* we are able to synchronize execution of neither commands issued to different command-queues nor of the host and a command. Therefore, OpenCL provides a *clFinish* command, which blocks the host until all previously enqueued OpenCL commands in an appropriate command-queue have finished. Therefore, future commands will not begin execution inasmuch as they will be neither submitted to command-queues until the previous ones have finished. The fact that this command is blocking makes it unsuitable when only execution of multiple commands need to be synchronized, since it also blocks a calling thread on the host despite it is not required.

To synchronize commands in different command-queues defined in the same context while not stalling a calling thread on the host we have to use so-called *event-objects*. An event object is generated when a command is submitted to a command-queue and communicates the status of the associated command. A *clEnqueueWaitForEvents* command, which accepts a collection of event-object as one of its parameters, ensures that all commands to which passed event-objects are associated have finished before any future commands queued to the command-queue begin execution. This means that the clEnqueueWaitForEvents command, in contrast to the clEnqueueBarrier, may be used to synchronize commands in

multiple command-queues and moreover allows to specify commands that have to complete before future commands can begin execution. State that OpenCL provides even a blocking counterpart of this command, namely *clWaitForEvents*, that may be used to synchronize the host execution with execution of commands forming a subset of previously submitted commands.

The last way how to synchronize execution of multiple commands in different command-queues defined in the same context is also based on utilizing event-objects. In this case, a collection of event-objects is passed to a command to be synchronized as one of its parameters. This command then can begin its execution only when each event-object in the passed collection communicates that the associated command has finished. This means that this way of synchronization provide much more fine-grained control than utilizing the clEnqueueWaitForEvents command, since this way constrains the execution order of a particular command and not of all future commands.

2.4 Memory Model

The OpenCL memory model defines 2 types of the memory objects: buffer objects and image objects, which are not usable for the purposes of this work, and therefore, will not be further described. A buffer object stores a one-dimensional collection of elements of a scalar data type, vector data type, or a user-defined structure. Elements of a buffer object can by accessed using a pointer by a kernel executing on a device.

The OpenCL memory model defines, besides memory objects, 5 distinct memory regions:

- **Host memory**: This memory region resides in the host RAM, and therefore, is visible only to the host. As in the case of the host program, OpenCL only defines how the host memory interacts with the OpenCL objects.

- **Global memory**: This memory region is accessible by the host, and permits reads and writes to all work-items in all work-groups. Reads and writes to the global memory may be cached depending on the capabilities of the device. This memory region and the host memory is in case a device is GPU, Xeon Phi or another coprocessor independent of each other. Due to this, data stored on the host needs to be transferred to the global memory or vice versa.

- **Constant memory**: This memory is a region of global memory that remains constant during the execution of a kernel. Thanks to this, this memory is better cached than the global memory.

- **Local memory**: This memory region is associated with a work-group and is accessible only by work-items in this work-group. The time needed to access variables in this memory region is guaranteed not to be greater than the time needed to access variables in global memory. Hence, a common optimization technique utilized in data-intensive applications is based on copying hot data to this memory region.
- **Private memory**: This memory belongs to a work-item, and therefore, is used to store its local variables.

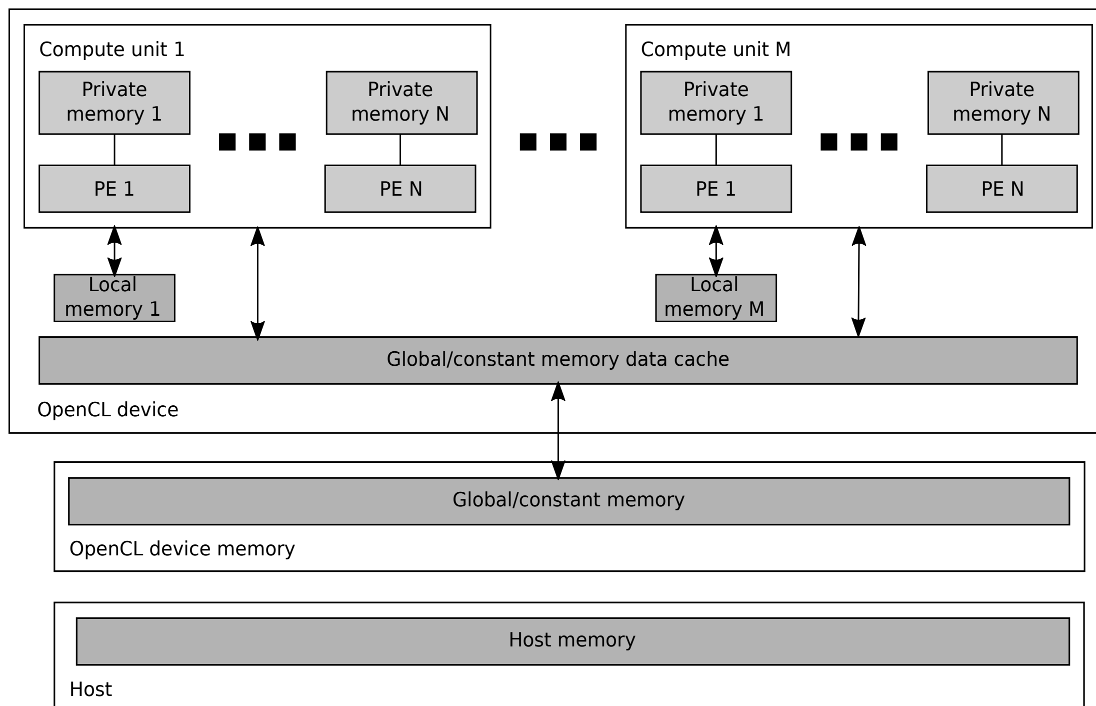

Figure 2.3: The mapping of the OpenCL memory model to the OpenCL platform model.

When we described the global memory, we stated that data has to be transferred between this memory and the host memory, unless the host and a device share memory and caches. Transferring of data can be performed in one of the two ways: by explicit **copying** of data or by **mapping** and **unmapping** regions of memory objects.

The mapping allows the the host to map a region of a memory object into its address space. Once the region from the memory object has been mapped, the host can access this region directly. The host has to unmap the region after use to propagate changes back to the devices.

In case of the device connected to the host by a bus, transferring data between the host memory and the global memory has to be done via this bus.

Consequently, such transferring is usually slower in order of units than accessing data in memory. This is determined by the fact that currently buses do not provide such throughput as memories. Hence, utilizing OpenCL is primarily convenient for applications in which the computation time is at least roughly equal to the time the host or the device wait for data being transferred.

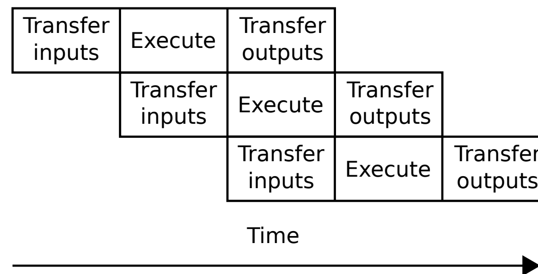

Figure 2.4: An example of transfers and executions pipelining.

However, some OpenCL devices support pipelining, i.e. simultaneous data transfers in both direction and computations, as illustrated in Figure 2.4. Therefore, the slowdown introduced by transferring data may be reduced by partitioning the input data into several parts to be processed independently. Such partitioning yields the following consequences:

- the time the device is waiting data is reduced since only a part of the original data is transferred
- the other parts of the original data may be transferred to the device while it executes on the preceding parts.

2.4.1 Accessing Shared Memory Simultaneously

When concurrent execution is introduced, a special attention has to be paid to the consistency of shared memory to avoid potential race conditions.

The first issue that must be addressed is concurrent manipulating the same memory location. For this purposes, OpenCL implements a few atomic operations, which only apply to integers and longs except for a single function for floats. These operations guarantee that an operation on a given memory location is thread-safe, which means no other work-item can access to that location while the atomic operation is executing.

Another issue to be addressed is the visibility of changes made by a single work-item to other work-items. This is essential especially in case when multiple work-items cooperate in computation. OpenCL guarantees that changes made by a work-item are visible to other work-items in the same work-group only at work-group synchronization point. For example, a work-group barrier forces all stores

defined before it to complete before any work-items in the work-group proceed past it. On the other hand, OpenCL does not provide any way of enforcing consistency of shared memory between work-items belonging to different work-groups.

2.5 Best Practices

To conclude this chapter we will present several programming rules and recommendations that are based on the previously stated facts.

2.5.1 Work Decomposition

The peak performance of an arbitrary device is possible to achieve only when all its computation resources are utilized. Therefore, a computation has to be partitioned to at least as many independent parts as the number of processing elements of a device this computation runs on, since otherwise some processing elements of the device will be idle. Therefore, devices equipped with up 1000 of processing elements, e.g., GPUs or other highly-parallel devices, necessitate partitioning of computations to be executed on them to an enormous number of independent parts.

2.5.2 Synchronization

Synchronization forces some processing elements to wait until a specific event occurs, e.g., all work-items in a work-group reach a synchronization point, or a command is communicated as finished, and therefore, it is expensive. Hence, synchronization should be avoided as much as possible.

2.5.3 Reusing of Data

Data should be kept on a device as long as possible. This means that programs that run multiple kernels on the same data should favor leaving the data on the device between kernel invocations, rather than reading intermediate results to the host and then sending them back to the device for subsequent calculations. Reusing of data thus improves the performance, since it reduces the number of transfers, which are fairly costly.

2.5.4 Data Layout

Every time when a computation executes on a collection, which will be further referred to as a source collection, of structures comprising multiple fields, a serious

attention should be paid to the selection of the data layouts in which the source collection will be arranged. Because this choice may have a large impact on overall computation performance. Although OpenCL supports an array of a structure we will arrange the source collection into an array of a simple data type, which we will be denoted as a destination array. There are 3 different data layouts that are being used to arrange such a collection into a destination array.

The first data layout is referred to as the Array-of-Structures (AoS). This data layout arranges fields of a single item of the source collection to consecutive indices of the destination array. However, using this data layout once multiple threads access the same field of consecutive items results in using of scatter and gather instructions, since these fields occupy indices with the stride sized the number of the fields of the structure. On this account, such memory access is termed as a strided memory access. Due to the hardware capabilities, scatter and gather instructions are typically less efficient than simple vector load and store instructions.

Another layout is called Structure-of-Arrays (SoA). In this case, consecutive indices of the destination array are occupied by the same field of consecutive items. Such access corresponds to a so-called coalesced memory access. Hence, using this data layout does not require usage of scatter and gather instructions in case when consecutive threads execute on consecutive elements of the source. On the other hand, this data layout suffers from poor spatial locality.

The last layout is referred to as Array-of-Structures-of-Arrays (AoSoA). This layout arranges items to an array of structures of small arrays. So, this layout provides a simple vector loads and stores, while does not suffer from poor spatial locality. The problem of AoSoA is readability of the code.

3. Intel Xeon Phi

Intel Xeon Phi Coprocessor is the brand name for all Intel Many Integrated Core (MIC) Architecture [7], based products. This architecture is targeted for highly parallel, High-Performance Computing workloads.

The Intel Xeon Phi coprocessor is connected to a CPU, which is referred to as a host, through a PCIe bus. This implies that data could be transferred between the host and the coprocessor with a nominal transfer speed 16 GB/s in each direction. This means that data transfers incur some overhead, and therefore, one should reduce data transfers as much as possible.

A single host system can contain multiple Intel Xeon Phi devices, which can communicate with each other through PCIe peer-to-peer interconnect without any intervention from the host.

3.1 Architecture

The Intel Xeon Phi coprocessor is composed up to 61 cores, each of which is derived from an Intel Pentium core, augmented by Intel 64 ISA, 4-way simultaneous multithreading, i.e. it can execute instructions from 4 hardware threads simultaneously, and a powerful vector processing unit (VPU). These cores are all connected through on-die wire interconnect, as shown in Figure 3.1.

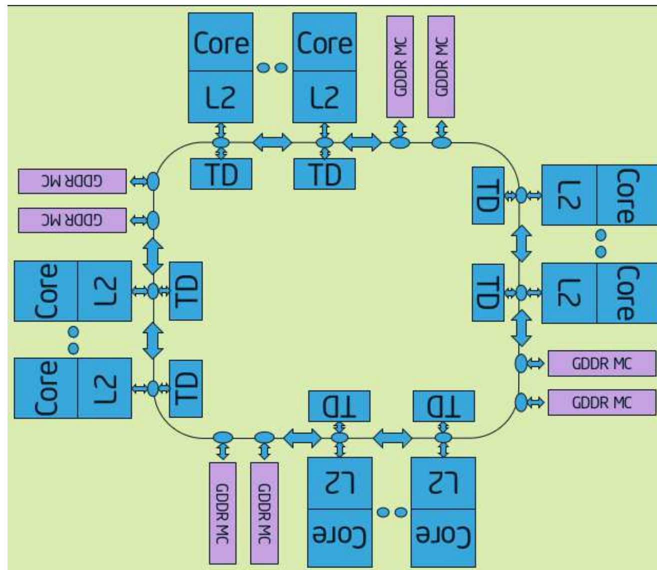

Figure 3.1: The Intel Xeon Phi processor microarchitecture [8]

.

The VPU, which is 512-bit wide, is fully pipelined and can execute most instructions with 4-cycle latency and single-cycle throughput. The cores do not

support any previous Intel SIMD extensions like MME, SSE, or AVX. Instead of these, a novel instruction architecture set is implemented to utilize the VPU.

The interconnection is implemented as a bi-directional ring, which carries data and instructions to various agents including cores, memory controllers providing a direct interface to GDDR5 memory on the device, and a globally distributed tag directory. These agents are connected to the ring through ringstops.

Each direction comprises three independent rings, which carry data, addresses, and acknowledgements. The data ring is 64 bytes wide, so it permits transferring data occupying a single cache line, which is also 64 bytes wide, simultaneously. The second ring is used to send read/write commands and memory address, and thus, is denoted as the address ring. The last one transfers flow control and coherence messages.

3.1.1 Vector Processing Units

A key hardware feature that dictates the performance of highly-parallel computing on Intel Xeon Phi is the VPU working on 512 bits, i.e. it can execute 16 single-precision or 8 double-precision operations at a time. It can execute most vector operations with four cycle latency and provides the maximal throughput of 1 instruction per cycle, what corresponds to the facts that the VPU is fully pipelined and the core is able to execute instructions from up to 4 hardware threads simultaneously. The VPU can read/write one vector per cycle from/to the vector registers file or data cache simultaneously with one vector operation.

Each VPU has 128 512-bit vector registers divided among the threads, thus, each thread is provided by 32 registers. Besides these vector registers, there are 16 mask registers per thread, which are part of the vector register file.

Vector Mask Registers

Employing the mask registers one can make the update of the target register element conditional on the bit content of a vector mask register, otherwise, all the destination elements are updated. For this reasons, these registers help to vectorize short conditional branches, as illustrated in Figure 3.2, where elements in *m1* register are values to which a predicate evaluates. For this reason, only the *v3* register elements corresponding to 1 bit in the *m1* register get updated.

VPU Pipeline

Each VPU instruction passes through one or more of the following pipelines to completion:

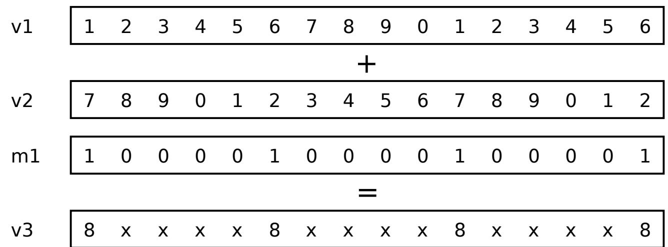

Figure 3.2: An example of conditional updating (x represents an unchanged value).

- **Double-precision pipeline**: Used to execute float64 arithmetic, conversion from float64 to float32, DP-compare instructions.
- **Single-precision pipeline**: Executes most of the instructions including 64-bit integer loads, float32/int32 arithmetic and logical operations, shuffle/broadcast etc.
- **Mask pipeline**: Executes mask instructions with one-cycle latencies.
- **Store pipeline**: Executes the vector store operations.
- **Scatter/gather pipeline**: Executes the vector register read/writes from sparse memory locations.

It should be noted that interleaving of pipelines and execution of dependent instructions incur time penalties. For this reason, the pipeline can throughput one instruction per cycle only when two independent SP or DP instructions are being executed.

3.1.2 Memory Architecture

The memory architecture of Intel Xeon Phi coprocessors, which is depicted in Figure 3.3, resembles the memory architecture of CPUs. This means that there is multilevel, hardware managed, coherent cache hierarchy on the top of the memory architecture.

Cache

Each core is equipped with a 32 kB L1 instruction cache and 32 kB L1 data cache and a 512 KB L2 cache. The L2 cache is inclusive of the L1 cache, that is, L1 cache lines have to be also included in the L2 cache. Unlike the L1 cache, the L2 cache is unified, so, it caches both data and instructions.

The coherency of data residing in both the L1 cache and the L2 cache among the cores on the ring is maintained by a *MESI* protocol. In addition to the *MESI*

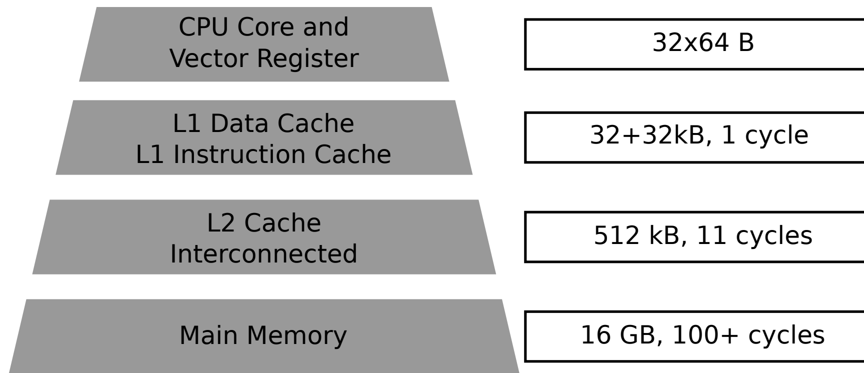

Figure 3.3: The memory architecture of the Intel Xeon Phi coprocessor.

protocol [11], the Intel Xeon Phi coprocessor implements a physically distributed tag directory TD along with a TD-based *globally owned, locally shared* (GOLS3) protocol. The reason for this is that the supplementing the *MESI* protocol with the *GOLS3* protocol locates the potential performance bottleneck of the *MESI*.

Tag Directory

The TD is attached to each core and gets an equal portion of the whole address space. Each physical address is uniquely mapped to TD through a reversible function. A TD entry, which is called a TD tag, contains the address, state and an ID of the owner of the cache line.

On an L2 cache miss, a core references a TD that is uniquely determined by the address, and therefore, it is not necessarily located on the core that generates the miss.

The TD is also responsible for initiating communication with the memory controllers. This communication is initiated on an L2 cache miss when a request is sent to a memory controller from the TD maintaining the requested cache line. Once the memory controller retrieves the requested cache line, it is returned to the requesting core.

State that the L1 cache has lower latency than the L2 cache, which has lower latency than the memory. For this reason, reusing of data residing as close as possible to the core is essential to achieving good performance.

Memory Controller

The Intel Xeon Phi coprocessor contains up to 8 memory controllers, which are evenly interleaved around the ring. Each of them has 2 channels, which communicate with GDDR5 memory at 5,5 GT/s. This is, the aggregate memory bandwidth equals to 352 GB/s.

The addresses are evenly distributed among the memory controllers to reduce

bottlenecks, and to provide optimal bandwidth. This implies that the consecutive memory locations are distributed among the memory modules and there is no way to place memory close to a core in order to provide optimal memory bandwidth.

Applicability of the Intel Xeon Phi Coprocessor

The utilization of the Intel Xeon Phi coprocessor could be beneficial only when the code to be run on the coprocessor has the following characteristics that fit the Intel Xeon Phi architecture:

- The code utilizes all available cores without keeping them idle and even scales with the number of available cores.
- The code is vectorizable and thus utilizes the VPUs efficiently.
- The communication with the host is minimized and overlapped with the computation as much as possible.

3.2 OpenCL Programming

As stated in Chapter 2, OpenCL is a framework for cross-platform, parallel programming. Although OpenCL applications may run on various platforms, their performance may vary on these platforms. This is caused by the fact that these platforms may have rather different HW design, and therefore, benefit from different application optimizations. For this reason, the remainder of this chapter will be devoted to the description of optimizations specific for the Intel Xeon Phi coprocessors.

3.2.1 Kernel Execution on Intel Xeon Phi Coprocessor

At initialization time, OpenCL creates as many software threads as the Intel Xeon Phi coprocessor contains logical cores, i.e. OpenCL creates 244 software threads in case the coprocessor comprises 61 cores, and pins each of them to one logical core.

After that, when a kernel is submitted for execution, each work-group is assigned to one software thread, which executes all work-items in this work-group. Combining this fact together with the fact that software threads are pinned to logical cores yields that logical cores correspond to compute units.

Parallelism among Work-Groups

Since different work-groups are executed by different logical cores they may execute in parallel. This means that utilizing all logical cores at a time necessitates at least as many work-groups as the number of logical cores, otherwise, some of the logical cores will not be utilized. Generally, employing a larger number of work-groups results in more flexibility in scheduling.

Despite the previous recommendation, the execution of work-groups should take at least 100 000 clock cycles in order to keep the proportion of thread switching overhead to actual work reasonably small.

Work-Group Level Parallelism

OpenCL on the Intel Xeon Phi coprocessor achieves parallelism also at a work-group level by executing a vectorized kernel, since such a kernel may be partly executed by the VPU. A kernel vectorization, which is done automatically by an implicit vectorization module[1], consists of unrolling a routine executing a work-group. Such an optimized routine is depicted in Listing 3.1.

```
1 kernel_wrapper(...)
2   #pragma unroll 16
3   for (int local_id = 0; i < WORK_GROUP_SIZE; ++local_id)
4     kernel_body(...)
```

Listing 3.1: Pseudocode of an optimized routine executing a work-group.

Although the routine executing a work-group comprises up to 3 loops, what corresponds to the maximal supported dimension of index spaces 2.3.1, only the innermost loop that corresponds to the first dimension of the NDRange may be unrolled.

Implicit Vectorization

The implicit vectorization module automatically vectorizes the work-group routine in the innermost loop, i.e., the code is unrolled by the vector size, which is 16 regardless of the type of data used in the kernel [10].

Nevertheless, the vectorized kernel is used only if the local size of the first dimension is not less than 16. Otherwise, the OpenCL runtime runs the scalar kernel for each of the work-items. This may introduce a significant performance penalty for work-groups having a small size in the first dimension. Therefore, it is recommended to round up the size in the first dimension to the closest multiple of 16.

[1] An implicit vectorization module is a part of the program build process.

Branch Statements

We distinguish two kinds of control flows, namely uniform and non-uniform control flow. A branch is uniform if it is guaranteed that all work-items within a single work-group will execute the same block of this branch. In the context of the Intel Xeon Phi coprocessor, that is, the branch predicate has to be constant.

```
1 uint mask = get_mask();
2 int res_if = if_block();
3 int res_else = else_block();
4 int res = (res_if & mask) | (res_else & not(mask));
```

Listing 3.2: An example of the masked execution.

In the latter case, a branch is non-uniform. Non-uniform control flows have worse performance than uniform ones since both blocks of the branches have to be executed using masked execution. An example of such execution is contained in Listing 3.2.

3.2.2 Local Memory

For the Intel Xeon Phi coprocessors, all OpenCL memory objects are implicitly cached by the hardware. Hence, the commonly used optimization technique based on employing the local memory for caching hot data is beneficial only in case of data scattered in the global memory but compacted in the local memory. In the other case, employing the local memory for caching hot data only introduces unnecessary overhead caused by redundant data copy and management. In other words, utilizing locality of data provides effective memory access on Intel Xeon Phi coprocessors. achieving the peak performance.

3.2.3 Comparison with GPUs

In conclusion of this chapter, we will briefly compare OpenCL programming for Intel Xeon Phi coprocessors with OpenCL programming for GPUs. The reason why we opted for GPUs and not for another kind of OpenCL devices is based on the fact that GPUs are suited for data-intensive computations just like Intel Xeon Phi coprocessors and moreover OpenCL programmers are typically familiar with programming for them.

Both Intel Xeon Phi coprocessors and GPUs have many cores and data to execute on kernels have to be transferred to or from them. Therefore, there are aspects that improve performance of OpenCL applications both on these coprocessors. These aspects include:

- Include enough work-groups, which themselves comprise enough work-items, within each NDRange.
- Employ consecutive data accesses or at least good data locality.
- Reduce data transfers from and to the host.
- Overlap data transfers and computations.
- Reduce synchronization on any level and usage of atomic operations.

On the other hand, there are aspects that improve performance of OpenCL applications on the Intel Xeon Phi coprocessors but not on GPUs:

- Avoid small work-groups, i.e. those whose execution takes significantly less than 100 000 clock cycles.
- Do not use local memory for caching data that occupy a block of global memory.

4. AT Model

Algorithmic trading uses a well-defined set of rules based on timing, price, quantity, and other mathematical models for identifying favorable opportunities. Such way of identifying favorable opportunities is also known as a *market analysis*.

We distinguish two kinds of the market analysis: a *fundamental analysis* and a *technical analysis*.

Fundamental Analysis The fundamental analysis uses fundamental factors related to an underlying asset in question to predict its future price. These factors can be grouped into two categories [12]:

- **Quantitative**: factors capable of being measured or expressed in numerical terms,
- **Qualitative**: factors related to or based on the quality or character of something, often as opposed to its size or quantity.

In fact, the fundamental analysis is more based on qualitative factors. Beside the fact that these factors are hard to be analysed automatically, they also have slow update frequency as most these factors are linked to quarterly reports. This, combined with the time needed to perform a thorough fundamental analysis, makes this analysis convenient predominantly for trades lasting at least weeks.

From the previous paragraph we conclude that the fundamental analysis is not suitable for HFT, and hence, it will not be described here any further. Instead of this, we will shift our attention to the thorough description of the technical analysis, which, unlike to the fundamental analysis, is convenient for HFT.

4.1 Technical Analysis

The technical analysis uses only historical data of trading instruments, such as previous prices and volumes, to predict the future price of the trading instrument. The prediction of the price is also referred to as financial time series forecasting.

Financial time series are inherently noisy and non-stationary, as introduced by Yaser [13]. These facts outline that there is no complete information that could be obtained from the previous behaviour of financial markets to fully capture the dependency between the future price and historical data. Due to this, quite sophisticated approaches have to be used for forecasting of financial time series.

4.1.1 Technical Indicators

To smooth out financial time series, technical indicators, e.g., *simple moving average*, *weighted moving average*, or *exponential moving average*, that are derived from historical data are used. The mentioned indicators are defined as follows, where p denotes all data of a single time series, p_t denotes the latest, i.e. current value of this series, and p_{t-i} denotes the i-th newest value.

Simple Moving Average

A simple moving average (SMA) is calculated, as its name suggests, as the average value over a defined number of time periods. The defined number of time is often called the order of the model and will be denoted by τ.

$$SMA(p, \tau) = \frac{p_t + \ldots + p_{t-\tau}}{\tau}$$

A weighted moving average (WMA) is derived from the SMA by assigning weights to items. Therefore, a WMA can emphasize recent data rather than old data.

Exponential Moving Average

An exponential moving average (EMA) decreases the weighting of data exponentially with its age. It weights new data with $1/\tau$ and old data, i.e. the previous *EMA*, with the remainder $(\tau - 1)/\tau$. This implies, the weighting of any item never reaches 0, and therefore the effect of any item is never entirely removed.

$$EMA(p_t, \tau) = \frac{p_t}{\tau} + \frac{\tau - 1}{\tau} EMA(p_{t-1}, \tau)$$

Each model used for financial time series forecasting may utilize one or more these indicators. The utilized indicators determine into which of two disjoint categories is the model classified:

- *Univariate analysis* uses only indicators restricted to the time series being predicted as the input variables,
- *Multivariate analysis* uses any indicators as the input variable.

4.1.2 Univariate Analysis

General, commonly used univariate models are based on AutoRegressive Integrated Moving Average (ARIMA) model. As the name suggests, this model combines an autoregressive model and a moving average model to predict future trends. The autoregressive model specifies that the output variable depends linearly on its own previous values and on a stochastic, i.e. imperfectly predictable, term.

ARIMA is based on the assumption that the time series are linear and stationary[1]. Regrettably, these assumptions are not compliant with the characteristics of financial time series, as stated in [14].

4.1.3 Multivariate Analysis

To model non-linear behavior, non-linear time series models, such as a Threshold Autoregressive model [15], an Autoregressive Conditional Heteroscedastic model [16], have been developed. These models and all models that will be introduced in the rest of this chapter perform the multivariate analysis since their inputs are not restricted to indicators derived from the time series being predicted.

However, the non-linear time series models are still limited in the sense of that these models are explicitly defined pursuant to the knowledge of relationships among underlying assets just as the linear time series models are. However, acquirement of the complete knowledge of these relationships is quite difficult, since there are too many non-linear patterns that should be taken into account. Consequently, these models may not be general enough to capture all important relationships.

Both linear and non-linear time series models are representatives of so-called model-base methods of predicting time series. From facts stated in previous paragraphs, it is obvious that these methods are not adequate for problems, whose solution requires knowledge difficult to obtain. Nevertheless, if we have enough historical data we can utilize so-called data-driven models for solving these problems.

The data-driven methods require only a few assumptions about models for problems to be predicted. A data-driven method corresponds to a function that is inferred from a given set of historical data, which are also called training data. State that this function inferring is known as a supervised learning task in the context of machine learning, and thus, the data-driven methods may be also referred to as supervised learning models. Construction of a supervised learning model based on the training data is being performed by an associated supervised learning algorithm.

4.1.4 Supervised Learning Algorithms

Supervised learning algorithms take a set of N training data $T = \{(\vec{x_i}, y_i)\}_{i=1}^{N}$, such that $\vec{x_i} \in \mathbb{R}^D$ is the vector of features, in our case of indicators, of the i-th input and $y_i \in \mathbb{R}$ is its label, i.e. the desired value. Based on this training set

[1]A time series is stationary if its statistical properties are all constant over time.

these algorithms seek a function g that is an element of a set of possible functions $G \subset \mathbb{R}^D \rightarrow \mathbb{R}$.

There are two basic approaches for choosing the desired function g: *Empirical Risk Minimization* and *Structural risk Minimization*. Both these approaches use a loss function $L : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ in order to measure how well functions fit the training data. For each training data $(\vec{x_i}, y_i)$ the loss of predicting $\bar{y_i} = g(\vec{x_i})$ equals to $L(y_i, \bar{y_i})$. This function is utilized by a risk function $R_T : G \rightarrow \mathbb{R}$ that is defined as the expected loss of g.

$$R_T(g) = \frac{1}{|T|} \sum_{(\vec{x_1}, y_1) \in T} L(y_i, g(\vec{x_i}))$$

Empirical Risk Minimization

In *empirical risk minimization*, the associated algorithm seeks a function g that minimizes the risk functions, that is, the supervised learning algorithm prioritizes functions that fit well to the training data but do not take into account their ability of generalization. Note that functions well fitting the training data and having poor generalization are denoted as overfitted. An example of such function is depicted in Figure 4.1, where the dots represent training data, the solid line represents the true function, and the dotted line denotes this unction.

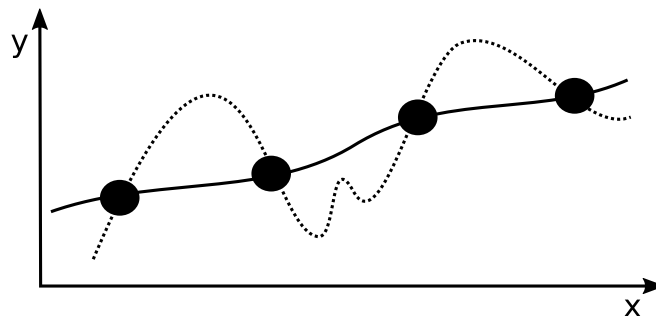

Figure 4.1: An example of overfitting.

Structural Risk Minimization

Structural risk minimization [17] seeks a function g that minimizes the upper bound of the generalization error rather than training error. This is done by incorporating a regulation penalty that prefers simple functions over complex ones into optimization.

Overfitting

Overfitting generally occurs when a model is excessively complex, such as having too many parameters relative to the number of the training data. Such a model usually has a poor level of generalization, i.e. ability to predict the correct

output values for data that were not seen during training, since it describes not only underlying relationships but also random errors and noise. In other words, overfitting occurs when a model begins to memorize the training data rather than generalize from the trend.

4.2 Support Vector Machine

A Support Vector Machine (SVM), which was proposed by Vapnik [18], is supervised learning model used for classification and regression analysis.

Although SVMs may be utilized for multiclass classification we will, for sake of simplicity, restrict our description only to SVMs executing binary classification.

A SVM model of binary classification represents a set of N independent and identically distributed training data $T = \{(\vec{x_i}, y_i)\}_{i=1}^{N} \subset \mathbb{R}^D \times \{-1, 1\}$ by a hyperplane separating entries of training data belonging one category, i.e. $\{\vec{x_i} \mid (\vec{x_i}, y_i) \in T \wedge y_i = -1\}$ from entries belonging to the other category and withal has the maximal largest distance to the nearest entries of both classes. Such a hyperplane is in Figure 4.2 denoted by H_2 whereas white dots represent entries of the training data belonging to one category, black dots represent entries of the training data belonging to the other category. We will denote a space containing all points distant from the separating hyperplane maximally as the closest points in the training data as a gap. Margins of this gap, which are denoted by g_1, g_2 in that figure, will be referred to as *gutters*.

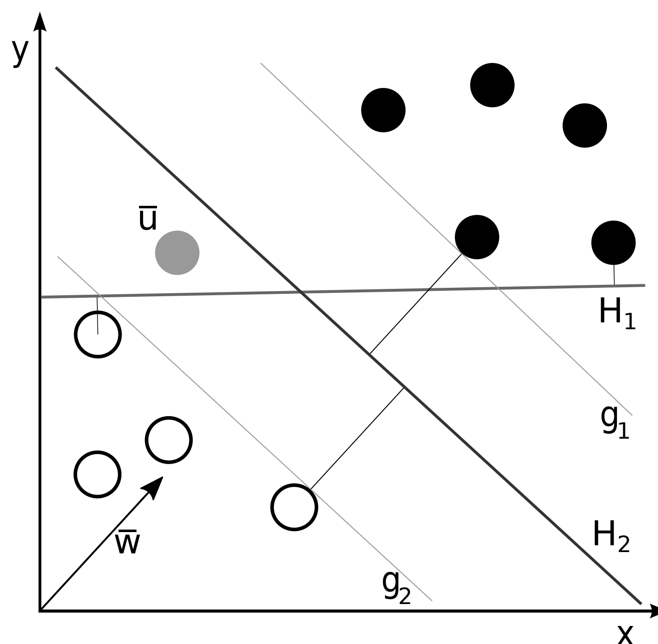

Figure 4.2: An instance of SVM.

The separating hyperplane splits the vector space of the training data to two

nonoverlapping parts. These parts are used for classifying new data since they are predicted to belong to one category pursuant to which of these parts includes it.

Maximizing the gap conforms to the fact that the SVMs are based on *structural risk minimization*, unlike most of supervised machine learning models that are based on *empirical risk minimization*. Thanks to this, SVMs are less prone to overfitting and thus provide good generalization with a much bigger probability than traditional models.

4.2.1 Theory of SVMs

SVMs carry out binary classification of data $\vec{u} \in \mathbb{R}^D$ pursuant to a decision rule $f_T : \mathbb{R}^D \to \{-1, 1\}$. The form of this rule will be expressed in terms of a vector $\vec{v} \in \mathbb{R}^D$ and a scalar value $o \in \mathbb{R}$. The vector $\vec{v}$ represents a so-called decision vector that has to be perpendicular to the separating hyperplane and the o corresponds to the offset of the separating hyperplane from the origin of the coordinate system in the direction of the vector $\vec{v}$. In other words, the offset allows classifying data that are separable by a hyperplane that does not pass the origin inasmuch as it moves the separating hyperplane in a direction of the decision vector. Now, the decision rule should be defined as follows:[2]

$$f_T(\vec{u}) = \begin{cases} 1 & \text{if } \vec{v} \cdot \vec{u} + o \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{4.1}$$

In the rest of this subsection, we will describe how $\vec{v}$ and o are defined based on the training data. The main outcome of this description will be the fact that both $\vec{v}$ and o may be expressed as a linear combination of the training data, and therefore, even the decision rule may be rewritten as a linear combination of the training data. This also means that the decision rule is easy to calculate using the training data and the scalar product.

Firstly, we have to define additional constraints for each entry of the training data. This constraint has the following form for each $i \in \{1, \ldots, N\}$:

$$y_i(\vec{v} \cdot \vec{x_i} + o) - 1 \geq 0 \tag{4.2}$$

Furthermore, we constrain the left side of the preceding equation to be equal to 0 only if $\vec{x_i}$ is in the gutters of the gap.

Now, the width of the gap w_T can be computed using two points $\vec{x_+}$, $\vec{x_-}$, each

[2]In all equations in this subsection $\cdot$ represents the scalar product.

lies in a different gutter, as:

$$w = (\vec{x_+} - \vec{x_-}) \cdot \frac{\vec{v}}{\parallel \vec{v} \parallel} = \frac{(\vec{x_+} \cdot \vec{v}) - (\vec{x_-} \cdot \vec{v})}{\parallel \vec{v} \parallel} = \frac{(1-o) - (-1-o)}{\parallel \vec{v} \parallel} = \frac{2}{\parallel \vec{v} \parallel} \quad (4.3)$$

In the preceding equation, the second equation utilizes distributivity of the scalar product. The third equation conforms to replacing $1 - b$ for $\vec{x_+} \cdot \vec{v}$ and $-1 - b$ for $\vec{x_+} \cdot \vec{v}$, where these operations are compliant with Equation (4.2) and with the previous constraint.

Note that SVMs maximize the width of the gap. However, maximizing $2/ \parallel \vec{v} \parallel$ is quite complicated, so we are going to convert this expression to a mathematically more convenient form.

$$max \frac{2}{\parallel \vec{v} \parallel} \rightsquigarrow max \frac{1}{\parallel \vec{v} \parallel} \rightsquigarrow min \parallel \vec{v} \parallel \rightsquigarrow min \frac{1}{2} \parallel \vec{v} \parallel^2 \quad (4.4)$$

Maximization of the gap with respect to the constraints defined in Equation (4.2) conforms to solving a so-called quadratic programming problem. For solving this problem we will incorporate the Lagrangian dual function, which yields the following dual formulation:

$$L = \frac{1}{2} \parallel \vec{v} \parallel^2 - \sum_{i=1}^{N} \alpha_i (y_i (\vec{v} \cdot \vec{x_i} + o) - 1) \quad (4.5)$$

where $\{\alpha_i\}_{i=1}^{N}$ are Lagrange multipliers.

By computing the first derivation of that expression with respect both to $\vec{v}$ and o and setting these expressions equal to 0 we can calculate the desired extrema.

$$\frac{\partial\ L}{\partial \vec{v}} = \vec{v} - \sum_{i=1}^{N} \alpha_i y_i \vec{x_i} = 0 \Rightarrow \vec{v} = \sum_{i=1}^{N} \alpha_i y_i \vec{x_i} \quad (4.6)$$

We just proved that the decision vector $\vec{v}$ is a linear combination of a subset of the training data. This subset, which will be referred to as T_S in the rest of this chapter, comprises only the training data $\vec{v}$ associated α_i that do not equal to 0. Entries included in T_S are called support vectors.

$$\frac{\partial\ L}{\partial\ o} = - \sum_{i=1}^{N} \alpha_i y_i = 0 \Rightarrow \sum_{i=1}^{N} \alpha_i y_i = 0 \quad (4.7)$$

Substitution of $\vec{v}$ and $\sum_{i=1}^{N} \alpha_i y_i$ with their expressions stated in 4.6 and 4.7, conduct equation 4.5 to the following form:

$$
\begin{aligned}
L =& \frac{1}{2}(\sum_{i=1}^{N} \alpha_i y_i \vec{x_i}) \cdot (\sum_{j=1}^{N} \alpha_j y_j \vec{x_j}) - \sum_{i=1}^{N} \alpha_i y_i ((\sum_{j=1}^{N} \alpha_j y_j \vec{x_j}) \cdot \vec{x_i} + o) - 1) \\
L =& \frac{1}{2}(\sum_{i=1}^{N} \alpha_i y_i \vec{x_i}) \cdot (\sum_{j=1}^{N} \alpha_j y_j \vec{x_j}) - \sum_{i=1}^{N} \alpha_i y_i (\sum_{j=1}^{N} \alpha_j y_j \vec{x_j}) \cdot \vec{x_i} - \sum_{i=1}^{N} \alpha_i y_i o + \sum_{i=1}^{N} \alpha_i \\
L =& \frac{1}{2}(\sum_{i=1}^{N} \alpha_i y_i \vec{x_i}) \cdot (\sum_{j=1}^{N} \alpha_j y_j \vec{x_j}) - (\sum_{i=1}^{N} \alpha_i y_i \vec{x_i}) \cdot (\sum_{j=1}^{N} \alpha_j y_j \vec{x_j}) - o \sum_{i=1}^{N} \alpha_i y_i + \sum_{i=1}^{N} \alpha_i \\
L =& \sum_{i=1}^{N} \alpha_i - \frac{1}{2}(\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \vec{x_i} \cdot \ \vec{x_j}).
\end{aligned}
\tag{4.8}
$$

Solving the preceding equation falls into the theory of mathematical analysis and so it is out of the scope of this work. We reached this point in order to prove that for solving this equation the only operation that is to be performed on the training data is required, namely the scalar product. This fact outlines that solving this equation is computationally manageable since the scalar product is easy to compute.

In addition the fact that the decision vector $\vec{v}$ is a linear combination of the T_S subset of the training data, let us rewrite the decision rule (4.1) as follows:

$$
f_T(\vec{u}) = \begin{cases} 1 & \text{if } \sum_{\vec{x_i} \in T_S} \alpha_i y_i \vec{x_i} \cdot \vec{u} + o \geq 0 \\ 0 & \text{otherwise} \end{cases}
\tag{4.9}
$$

This means that even the decision rule depends only on the scalar product, so, we just proved the second part of our statement that was mentioned at the very beginning of this subsection.

Until now, we have tacitly assumed that the training data are linearly separable. However, SVMs are also capable of performing classification of non-linearly separable data. This is done by utilizing a so-called kernel function.

4.2.2 Kernel Function

To separate non-linearly separable input data, these data are mapped to a higher dimensional space, in which they are linearly separable. The mapping is done using a function $\phi : \mathbb{R}^D \rightarrow \mathbb{R}^F$, where $F > D$. An example of such mapping is depicted in Figure 4.3, where the outline box depicts the separation hyperplane. The target vector space is called a feature space and image of input vector is known as a feature vector, in the context of SVMs.

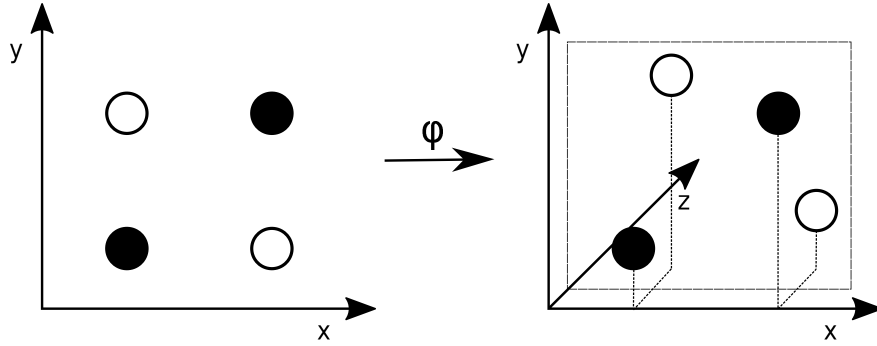

Figure 4.3: An example of mapping of non-separable inputs to a feature space.

However, we proved that we do not need to compute the image of a new data in order to classify it but rather compute the scalar product between this image and the images of all support vectors. That is, we seek a function $K : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}$ conforming $K(\vec{x_i}, \vec{x_j}) = \phi(\vec{x_i}) \cdot \phi(\vec{x_j})$ for any pair of $(\vec{x_i}, \vec{x_j}) \in \mathbb{R}^D \times \mathbb{R}^D$. Such a function is called a kernel function and its utilization transforms the decision rule to the following form:

$$f_T(\vec{u}) = \begin{cases} 1 & \text{if } \sum_{\vec{x_i} \in T_S} \alpha_i y_i K(\vec{x_i}, \vec{u}) + o \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

Note that machine learning methods utilizing the kernel functions are called kernel methods. More information about the kernel methods and constraints the kernel methods have to fit are provided in Vapnik [18].

Recently, multiple kernel learning (MKL) methods have been proposed, as stated in Gönen [19]. In this case, we use multiple kernels instead of selecting one particular kernel and its parameters:

$$k_\eta(\vec{x_i}, \vec{x_j}) = f_\eta(\{k_p(\vec{x_i}, \vec{x_j}) | p \in \{1, \ldots, P\}\}) \tag{4.11}$$

where a combination of functions $f_\eta : \mathbb{R}^P \rightarrow \mathbb{R}$, can be any linear or nonlinear function.

There are two main cases of usage of MKL:

- Various kernels correspond to various notations of similarity and instead of trying to find which works best, a learning method does the picking or a combination of them for us.

- Different kernels may use inputs from various representations or even various sources. Hence, these inputs have various measures of similarity corresponding to various kernels. Therefore, combining kernels is one possible way of computing the overall similarity.

Up to now, we have described SVM in the context of binary classification, because it is easier to show how the desired function is derived from the training data than in the context of functions approximations. Since this work is not aimed to provide a deep description of the theory of functions approximations using SVMs we are going to explain this topic at a very high-level in the following subsection.

4.2.3 SVM in Regression

In 1996, Vapnik [20] proposed a version of SVM for regression, which is called Support Vector Regression (SVR) that is dedicated to approximating an unknown function.

Similarly to the binary classification, the approximation of unknown function is based on a subset of independent and identically distributed training data $T = \{(\vec{x_1}, d_1), \ldots, (\vec{x_N}, d_N)\} \subset R^D \times R$, where d_i represent the desired values. The SVRs approximate a function as follows:

$$f_T(\vec{u}) = \sum_{\vec{x_i} \in T_S} \alpha_i K(\vec{x_i}, \vec{u}) + o \tag{4.12}$$

Note that we followed notation introduced in Section 4.2.1 and Section 4.2.2, and thus, T_S denotes a subset of the support vectors, α_i denote the weights of the support vectors, and K resembles the kernel function.

Note that we can also put additional constraints to the desired function, such as the maximal deviation from the actually obtained targets d_i for all entries of the training data or restrict its complexity.

4.3 Existing AT Model

In the preceding parts of this chapter, we have laid the theoretical foundation for the description of an algorithmic trading model in question.

The model is based on multiple kernel SVR, that is, it provides quite a sophisticated approach to approximating financial time series. Such approach is compliant to the fact that correctness of decision sent to the market is essential.

Remind ourselves that this work is intended to inspect whether the evaluation of this model may be accelerated by employing highly parallel architectures. And since we are not concerned with the construction of this model, all variables forming the model are constants or randomly generated values.

In the following lines, the meaning of separate variables that are forming the model along with their particular values, when defined, will be provided. In a case

when we are not concerned with the form of kernels the model K_m, where the subscript m is used to identify a separate model, has the following form:

$$K_m(\vec{u}) = \beta_m + \sum_{s=1}^{S_m} \alpha_{m,s} \prod_{d=1}^{D_m} \left(\overline{K}_{m,d}(\vec{\gamma}_{m,s}, \vec{u}) + \delta_{m,d} \right) \tag{4.13}$$

Realize that this equation is an instance of Equation (4.12), since β_m corresponds to o, $\vec{\gamma}_{m,s}$ corresponds to $\vec{x_s}$, and K is substituted with the product of $\overline{K}_{m,d}$. This means that this model represents a multiple kernel model 4.2.2, which accepts the vector $\vec{u} \in \mathbb{R}^1 0$ as input.

The variable S_m represents the number of support vectors. The value of this variable equals 1000 for testing purposes. The variable D_m that denotes the number of the kernel methods to be aggregated equals 10 and the vectors $\vec{\gamma}_{m,s}$ encompass 10 elements, just like the input vector.

In order to define the form of the kernel methods $\overline{K}_{m,d}$, we have to define so-called transformation functions $t_{m,d}$ first. These functions restrict separate values of the input vectors, and therefore, may be seen as sort of preprocessing. In this work, only transformation functions having one of the following notations are allowed:

- *identity*, i.e.

$$t_{m,d}(x) = x.$$

- *restriction* to the constant value in one or both directions, i.e.

$$t_{m,d}(x) = \begin{cases} x & \text{for } x < a_{m,d} \\ a_{m,d} & \text{for } x \geq a_{m,d} \end{cases}$$

- *continuous restriction*, i.e.

$$t_{m,d}(x) = \begin{cases} \frac{a_{m,d}+b_{m,d}}{2} & \text{for } x \leq a_{m,d} \\ \frac{b_{m,d}^2}{2(b_{m,d}-a_{m,d})} - \frac{a_{m,d}}{b_{m,d}-a_{m,d}} x + \frac{x^2}{2(b_{m,d}-a_{m,d})} & \text{for } a_{m,d} < x \leq b_{m,d} \\ x & \text{for } b_{m,d} < x \leq c_{m,d} \\ \frac{c_{m,d}^2}{2(c_{m,d}-d_{m,d})} - \frac{d_{m,d}}{c_{m,d}-d_{m,d}} x + \frac{x^2}{2(c_{m,d}-d_{m,d})} & \text{for } c_{m,d} < x \leq d_{m,d} \\ \frac{c_{m,d}+d_{m,d}}{2} & \text{for } d_{m,d} \end{cases}$$

The variables $a_{m,d}$, $b_{m,d}$, $c_{m,d}$, $c_{m,d}$ represent constants of the kernel method. Now, we will take a look at the description of the kernel methods. Each kernel

method $\overline{K}_{m,d}$ conforms to the following notation:

$$\overline{K}_{m,d}(\gamma_1, \ldots, \gamma_{D_m}, u_1, \ldots, u_{D_m}) = f_{m,d}\left(\sum_{i \in S_{m,d} \subseteq \{1,\ldots,D_m\}} g_{m,d}(\gamma_i, t_{m,d}(u_i))\right) \tag{4.14}$$

The choice of the function $f_{m,d}$ directly determines the choice of the function $g_{m,d}$. In this work only following combinations of these functions are allowed:

- $f_{m,d}(y) = y$, $g_{m,d}(\gamma_i, y_i) = \frac{\gamma_i * y_i}{\sigma_{m,d,i}}$
- $f_{m,d}(y) = \frac{1}{1+y}$, $g_{m,d}(\gamma_i, y_i) = \frac{\gamma_i - {y_i}^2}{\sigma_{m,d,i}}$

The variable y_i accepted by the functions $g_{m,d}$ represents the value u_i transformed using the function $t_{m,d}$, i.e $y_i = t_{m,d}(u_i)$.

Note that function $g_{m,d}$ is performed on a subset S of all the kernel method parameters, that is, each kernel method considers only of some dimensions of the feature space. This subset contains from 1 up to 3 elements according to the following probabilities: $P(|S| = 1) = 0.5$, $P(|S| = 2) = 0.3$, and $P(|S| = 3) = 0.2$.

When we put all the previous facts together we get that the models have the following form:

$$\begin{aligned} &K_m(u_1, \ldots, u_{D_m}) = \\ &= \beta_m + \sum_{s=1}^{S_m}\left(\alpha_{m,s}\prod_{d=1}^{D_m}\left(f_{m,d}\left(\sum_{i \in S_{m,d} \subseteq \{1,\ldots,D_m\}} g_{m,d}\left(\gamma_{m,s,i}, t_{m,d}(u_i)\right)\right) + \delta_{m,d}\right)\right) \end{aligned} \tag{4.15}$$

From the previous equation it ensues that the evaluation of our model is primarily composed of three nested loops, where separated iterations of each of these loops are independent each other since they have no side effects. Consequently, the evaluation of our model on a single input might be parallelized on the level of any of these loops.

5. Implementation

The first part of this chapter is dedicated to the analysis of possible approaches to parallelization of the evaluation of our AT models on one or more inputs. Afterwards, an implementation for each of the considered architectures, namely a multi-core architecture (commodity CPUs), and a many-core architecture (Intel Xeon Phi), will be proposed based on this analysis. Most attention will be paid to an implementation utilizing the Intel Xeon Phi coprocessor using OpenCL. Due to this, each of these implementations will be tailored to the corresponding architecture, and thus, will maximally utilize features of this architecture.

Besides the above-mentioned, this chapter will provide the description of important architectural decisions, e.g. the choice of programming language and utilized libraries. To conclude this chapter, we will describe how our many-core implementation fit for another many-core device, namely GPUs, and what modifications would be required to fine-tune our implementation for this device.

5.1 Analysis

Parallelization of an arbitrary computation requires partitioning of this computation into tasks, i.e. units of work that can run concurrently. Although tasks yielded by such partitioning might not be necessarily independent of each other, the degree of dependency between them should be as small as possible. Forasmuch as dependencies between tasks introduce the need for synchronization that is expensive and might deteriorate the overall performance. Therefore, we will describe only approaches to partitioning the evaluation of one or more models on one or more inputs into tasks that exhibit a reasonably small amount of dependencies.

Partitioning of computations into tasks, which allows utilization of all logical cores, is necessarily but not sufficient for achieving the peak performance on contemporary hardware. In order to get the maximum from this hardware, vector operations have to be also employed during the tasks execution, since these operations provide higher throughput than scalar operations.

Before we list and analyze approaches exhibiting a reasonably small amount of dependencies, we will propose one optimization that substantially influences the amount of work assigned to a task for all these approaches.

5.1.1 Preprocessing

The kernel methods $\overline{K}_{m,d}$ perform transformation of their parameters by applying the functions that were denoted as $t_{m,d}$ in Section 4.3. These functions are executed with the same parameters within each iteration of the outermost loop, which iterates through support vectors, and therefore, it is valid to extract these transformations from these iterations, store yielded values to a new array and change the kernel methods to operate on this array. The array holding transformed values typically contains more elements than the array of original values, since kernel methods operate on multiple (2.1 in average) parameters.

Utilizing this preprocessing noticeably reduces the computational complexity of each iteration and thus even the computation complexity of the whole evaluation of one model on one input. In an average case, this processing reduces the work associated with the evaluation of one model on one input roughly by 30%.

The considered transformation functions have no side effects, and therefore, they may execute in parallel without any restrictions.

5.1.2 Approaches to Parallelization

As stated in Section 4.3, the evaluation of our model on a single input can be parallelized on the level of any of the nested loops: $\sum_{s=1}^{S_m}$, $\prod_{d=1}^{D_m}$, $\sum_{i \in S_{m,d} \subseteq \{1,\dots,D_m\}}$. Moreover, the evaluation of any model K_m has no side effects, so these models or even a single model on multiple inputs may evaluate in parallel. The following list summarizes the levels on which the evaluation of one or more models on one or more inputs may be parallelized.

1. the evaluation of models K_m on a collection of inputs,
2. the evaluation of iterations of the outermost loop $\sum_{s=1}^{S_m}$, which iterates the support vectors $\gamma_{m,s}$,
3. the evaluation of iterations of the loop $\prod_{d=1}^{D_m}$ iterating the kernel methods $\overline{K}_{m,d}$,
4. the evaluation of iterations of the innermost loop $\sum_{i \in S_{m,d} \subseteq \{1,\dots,D_m\}}$, i.e the loop iterating relevant elements of the array of transformed values.

Parallelization on the level of the evaluation of models K_m itself may be achieved by multiple approaches. The first of them conforms to assigning the evaluation of multiple models on multiple inputs to a task. This approach conforms to partitioning inputs into chunks and assigning the evaluation of each of these chunks to tasks that together encompass the evaluation of all models.

This approach yields quite large tasks since the evaluation of a single model on a single input takes approximately 500 000 instructions under conditions, e.g., the number of support vectors, the number of kernel methods, etc., defined in Section 4.3.

Another approach lays in assigning the evaluation of a single model on multiple inputs to a task. Although this approach might seem identical to the previous approach, it is preferable in terms of reusing data, since only the constants of a single model, e.g., an array holding the weights of support vectors α_m, or an array encompassing the support vectors themselves γ_m, are accessed during the task execution. Therefore, these constants can be loaded from memory only when evaluating the first input and hereafter reused from the cache. There is important to note that these constants are represented by approximately 11 000 values while a single input is represented by 10 values, therefore it is more convenient to reuse these constants than inputs.

Both the previous approaches could be modified in such a way that a single input would have been evaluated within a task. Nevertheless, these approaches would not provide any option for reusing the constants of models.

The following approach partitions the evaluation of a single model on a single input, which conforms to calling the function K_m depicted in Listing 5.1, on the level of the outermost loop, i.e the only loop shown in this listing. The function $\overline{K}_m$ depicted in this listing represents the multiple kernel method used by the model K_m.

```
1 public float Km(float[] transformed_input)
2 {
3   float result = βm;
4   for(int s = 0; s < Sm; ++s)
5   {
6     result += αm[s] * K̄m(γm[s], transformed_input);
7   }
8   return result;
9 }
```

Listing 5.1: A snippet of source code of the m-th model.

All iterations of the loop depicted in the preceding listing modify the shared variable *result*, which serves as an accumulator. Due to this, modifications of this accumulator would have to be done using atomic operations or in a critical section in order to preserve data consistency of this variable when multiple iterations would be executed in parallel. Nevertheless, utilization of both atomic operations and critical sections might deteriorate the improvement obtained by utilizing multiple computational units, and thus, is not recommended. Therefore, this function had to be modified in order to be efficiently parallelizable.

The mentioned modification laid in partitioning of this function into 2 phases when the first phase encompasses the evaluation of the function $\overline{K}_m$ on all the support vectors storing the yielded values to a new array, which will be further referred to as *tmp_results*. This means that this array at index s contains a value computed as $\alpha_m[s] * \overline{K}_m(\gamma_m[s], transformed_input)$. The second phase then includes aggregation of values stored in that array to form the result. The important fact is that both these parts are well parallelizable.

The function $\overline{K}_m$, has no side effects, and thus, can be evaluated on multiple support vectors simultaneously. This is, parallelism within the first phase can be achieved by assigning the evaluation of the function $\overline{K}_m$ on one or more support vectors to a task. This approach to achieving parallelism conforms to data parallelism since the same function is applied to diverse data concurrently. Due to this, this approach is suited primarily for architectures targeted for highly parallel workloads. Note that the evaluation of this method on a single support vector takes roughly 500 instructions under conditions defined in Section 4.3.

To parallelize the second phase, we may take advantage of the fact that the sum is a commutative and associative operation, hence, it does not require one particular order in which it has to be applied on the values. Thanks to this feature, we can partition the original collection into independent subcollections, compute partial sums of these subcollections in parallel, and then aggregate the partial sums to form the result. Such an approach is commonly referred to as a parallel reduction and will be more thoroughly described later in the section dealing with the many-core parallelization.

The function $\overline{K}_m$ is formed by the loop $\prod_{d=1}^{D_m}$ iterating the kernel methods $\overline{K}_{m,d}$, as stated in Equation (4.13). Therefore, the evaluation of this function can be parallelized in the same way as the evaluation of the function K_m. Nevertheless, such an approach would yield too small tasks, inasmuch as the evaluation of each of the kernel methods $\overline{K}_{m,d}$ takes maximally tens instructions. Besides, this approach would not correspond to the data parallelism, because different kernel methods $\overline{K}_{m,d}$ might differ in their form. Both mentioned facts yields that utilizing of this approach is suited for none of the considered architectures.

Although the loop $\prod_{d=1}^{D_m}$ is not suitable for parallelization it is well vectorizable, since each kernel method $\overline{K}_{m,d}$ evaluate the same function $g_{m,d}$ on up to three elements. In other words, the kernel method $\overline{K}_{m,d}$ may evaluate the method $g_{m,d}$ on multiple elements simultaneously using vector operations. This also means that even all the preceding approaches may utilize vector operations if the evaluation of the methods $\overline{K}_{m,d}$ is vectorized.

From above-stated facts we conclude that even the last stated approach to achieving parallelism, i.e. assigning the evaluation of one or more iteration of the

loop $\sum_{i \in S_{m,d} \subseteq \{1,\ldots,D_m\}}$ to a task would also yield too small tasks. Therefore, we will not discuss this approach in more depth.

5.2 Architecture Specific Parallelization

In this and the following section, we will propose an implementation for both of the considered parallel architectures. These architectures are equipped with different numbers of computational units, and therefore, benefit from various approaches to partitioning the evaluation of one or models on one or more inputs to tasks.

5.2.1 Multi-Core Parallelization

Multi-core architectures, which are represented by commodity CPUs, are equipped with up 10 cores, each of them comprises 1 or 2 hardware threads. This fact combined with the fact that utilizing computational unit introduces management overhead determines that these architectures are not tailored for execution of an enormous number of small tasks.

In Section 5.1, we proposed 2 approaches yielding quite large tasks. Both these approaches do not assign a part of the evaluation of one model on a single input to a task, but on the contrary, they assign the evaluation of one or more models on multiple inputs to a task. These approaches differ just by the number of models whose evaluation is assigned to a task. In the above-mentioned section, we also stated that the approach of assigning the evaluation one model on one or more inputs is much more cache-friendly than the other one is. Based on this fact we decided to use this approach to achieving concurrency on this architecture.

As it is also stated in the above-referenced section, the selected approach results in accessing only the constants of a single model within the task execution. That is, these constants have to be loaded from the memory only when the first input is evaluated and hereafter can be reused from the cache. This means that the more inputs are evaluated by single task the fewer data transfers are required.

Despite the previous assertion, assigning the evaluation of one model on all inputs, which would require the least data transfers, to a task is not a convenient approach. Because even though we assume that all the models should evaluate the same number of inputs, they might take different times to complete. This is based on the fact that diverse models different kernel methods, and therefore,take different times to evaluate. In other words, this approach is predisposed to result in an uneven decomposition of the work.

When we put together the facts stated in the previous paragraphs, we get

that assigning the evaluation of one model on multiple inputs to a task is the most convenient approach as it ensures that a unit of work assigned to a task is complex enough while preserving the possibility to distribute the total work into tasks evenly.

Data Parallelism

Different iterations of the outermost loop $\sum_{s=1}^{S_m}$ might be executed simultaneously using data parallelism, as stated in Section 5.1. Nevertheless, utilization of data parallelism on the considered architectures using the selected programming language, whose choice will be discussed in Section 5.4, is quite challenging, forasmuch as it requires writing of explicitly vectorized code [21]. Therefore, we did not utilize data parallelism on these architectures.

5.3 Many-Core Parallelization

Many-core architectures, which are in our case represented by the Intel Xeon Phi coprocessor, have hundreds of logical cores and thus are equipped with significantly more computational units than multi-core architectures. Consequently, these architectures require partitioning of work into much more tasks than multi-core architectures do for utilizing all their computational units. On the other hand, a unit of work assigned to a single computational unit still has to be complex enough so that the improvement obtained by its utilization surpass its management overhead.

Related Works

Although SVMs are commonly used for solving classification/regression problems, we found only two works [23], [24] dealing with parallelization of the classification/estimation phase. Both these works partition the evaluation of a single input into two phases, as described in Section 5.1. These works moreover assume the kernel method has the form of the vector multiplication, and therefore, the first phase, within which the kernel method executes on all support vectors and yielded values are stored to a new array for further processing, can be parallelized using the vendor specific *Single precision floating General Matrix Multiply* routine. However, the form of our kernel methods does not conform to the vector multiplication, and consequently, we had to propose our own approach to achieving parallelism within this phase.

Catanzaro, Sundaram, and Keutzer [23] also state that in the second phase, which was also defined in the above-referenced section, they aggregate values

yielded within the first phase using parallel reduction.

5.3.1 Evaluating Kernel Method In Parallel

In Section 5.1, we proposed several approaches to parallelizing the evaluation of one or more models on one or more inputs, nevertheless, the only one of these yields task conforming to data parallelism. Hence, we decided to adopt this approach as it seemed to be the best suited for these architectures.

The selected approach utilizes partitioning of the evaluation of a model on one input into two phases. In the first phase, whose implementation will be discussed in this and the following subsections, the multiple kernel method $\overline{K}_m$ is evaluated on all the support vectors and yielded values are stored to the array *tmp_results*, which resides in the global memory. Values stored in this array are thereafter aggregated within the second phase, whose implementation will be discussed in a standalone subsection.

In the above-referenced section, it was also stated that parallelism within the first phase can be achieved by assigning the evaluation of $\overline{K}_m$ on one or more support vector to a task, which is referred to as a work-item in the context of OpenCL, as stated in Section 2.3.1. We decided to assign the evaluation of the kernel method on a single support vector to a work-item, since this approach leads to utilization of more work-items, and thus even to utilization of more work-groups, than approaches assigning the evaluation of the kernel method on multiple support vectors to a task.

The preceding decision was based on recommendation stated inSection 3.2.3 that the more work-groups participate within a kernel invocation the better. When this recommendation reflects the fact that different work-groups execute on different logical cores. Hence, utilization of more work-groups results in utilizing more logical cores, and thus, even in more efficient utilization of the coprocessor.

The selected approach ensures parallelism among work-groups since multiple work-groups participate in the evaluation of the kernel method on all support vectors. Nevertheless, to achieve the peak performance the VPU, which was described in Section 3.1.1, providing parallelism at the work-group level has to be utilized as well. To utilize the VPU it is not needed to write explicitly vectorized code, because the OpenCL compiler contains implicit vectorization module, which automatically vectorizes scalar code yielding the vectorized kernel, as described in Section 3.2.1. This vectorized kernel is then used if the local size in the first dimension is divisible by 16. Because of this, we decided to round up the global index size, which conforms to the total number of work-items to be executed, to the closest multiple of 32 and set the size of work-groups to 32. This number

ensures that the mentioned condition is always fulfilled, and whilst yields a more convenient amount of work assigned to a work-group than it would have been provided by 16.

Even though the selected approach is efficient in terms of parallelization, it provides no opportunity to reuse the model constants (γ_m, α_m) within a single kernel invocation. Therefore, these constants, which are immutable within multiple kernel invocations, have to be always loaded from the constant memory. Nevertheless, accessing data residing in the constant/global memory is noticeably slower than accessing cached data, and therefore, reusing of data is favorable for achieving the peak performance, as stated in Section 3.1.2. Considering these facts, we adapted the work assigned to a single work-item in a way that it evaluates the same support vector against a collection of multiple consecutive inputs, which will be further referred to as a *block*. This modification converts the corresponding kernel to the form depicted in Listing 5.2. In this listing, the parameter $transformed_inputs$ holds transformed inputs arranged as Array-of-Structures (AoS), when this data layout was described in Section 2.5.4, IPB represents the total number of inputs to be evaluated, BS conforms to the number of work-items participating in the evaluation of one block, and $EPTI$ stands for the number of elements per a transformed input.

```
1 __kernel void K_m(constant float* transformed_inputs, global float*
      tmp_results)
2 {
3   int g_id = get_global_index(0);
4   for(int i = 0; i < IPB; ++i)
5   {
6     tmp_results[g_id + i * BS] = α_m[g_id] * K̄_m(γ_m[g_id],
      transformed_inputs[i* EPTI]);
7   }
8 }
```

Listing 5.2: Source code of the adapted kernel.

As noted in Section 3.1, Intel Xeon Phi coprocessor core is equipped with 32 kB L1 data cache and is capable of running 4 hardware threads to which work-groups are assigned. These facts combined with the fact that each work-item accesses 11 constant values, where 10 values represent a single support vector and the remaining value represents its weight that are not accessed by any other work-item in the same work-group gives that maximally 5632 $B = 4 \cdot 32 \cdot 44\ B$, where 32 conforms to the number of work-items in each work-group, are occupied by these constants during the execution of appropriate work-groups. Due to this, these constants along with units of constants that are accessed by all work-items in these work-groups fit to the L1 data cache altogether, and thus may be cached

during the whole kernel execution.

Moreover, this approach even provides a better ratio of the work assigned to a work-group to overhead of its management and management of associated data transfers than the original approach that evaluates a single input within a single kernel invocation.

5.3.2 Evaluating Multiple Blocks Simultaneously

Using the previous approach, only a subset of compute units, whose cardinality equals to $BS/32$, is utilized in a single kernel invocation. However, such utilization of the Intel Xeon Phi coprocessor does not comply with the recommendation that the number of work-groups should not be smaller than the number of compute units, as stated in Section 3.2.1. Based on this recommendation, we adopted an approach of evaluating multiple blocks in parallel within a single kernel invocation. This means that we define the global size of a kernel invocation as $BS \cdot NB$, where NB denotes the number of blocks that execute concurrently within a single kernel invocation, and assign the evaluation of the iteration $i = G_i \% BS$ of the block $b = G_i / BS$ to a work-item with the global ID G_i. Thanks to this, all compute units may be utilized by a single kernel invocation.

Since all inputs submitted for the evaluation are no more evaluated by a single block, their mapping to these block has to be specified. We decided for mapping consecutive inputs to a single block, as depicted in Figure 5.1, since such an approach provides better data locality than an approach of mapping consecutive inputs to consecutive blocks.

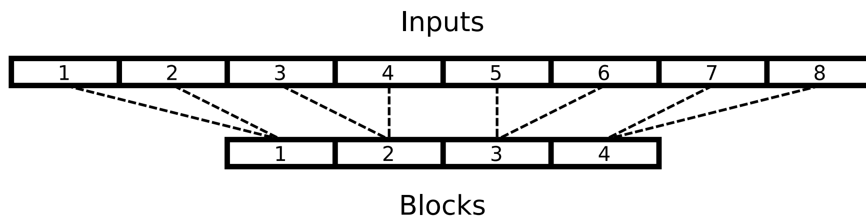

Figure 5.1: Mapping of inputs to blocks in case of 4 blocks where each of these blocks processes 2 inputs.

The execution of multiple blocks within a single kernel invocation is not the only way how to execute several blocks concurrently. Another way how to do it is submitting kernel execution commands to an out-of-order command-queue or to multiple command queues, as noted in Section 2.3.2. Nevertheless, execution of many small kernels is predisposed to provide worse performance than execution of fewer larger kernels, since all these kernels and associated data transfers necessitate the management.

5.3.3 Parallel Aggregation

In this section, we will follow up on the facts stated in Section 5.1 and describe how to aggregate the values yielded from the first phase in parallel using the reduction, which is a preferred way of aggregating values in highly-parallel environments[1].

Decomposition of a collection to be aggregated into independent subcollections may be depicted by a so-called reduction tree. Each reduction tree has a form of a rooted tree, where leaves represent separate items of the original collection and inner nodes on the level i constitute the aggregation of appropriate nodes on the level $i + 1$. This implies that there is no dependency between nodes on the same level, and therefore, computations represented by nodes on the same level may be executed concurrently.

There are both commutative and associative reduction trees, whose applicability is determined by the features of the aggregation function. And since the sum is both a commutative and associative operation, its reduction tree may be any of them.

Commutative Reduction Tree

Commutativity permits reordering of operands. Hence, inner nodes are not restricted to aggregate only consecutive nodes on the lower level. This may be used in such a way that consecutive work-items in a single work-group may access consecutive operands at the same time using vector load/store operations which provide better memory bandwidth than scalar load/store operations. Due to this, the commutative reduction, whose tree is depicted in Figure 5.2, is more appropriate for data parallel execution than the associative reduction, which utilizes just regrouping of its operands.

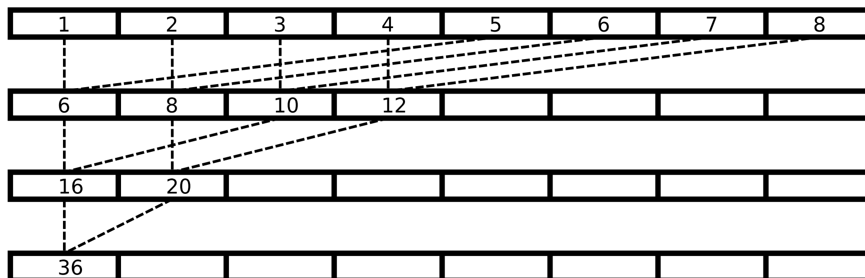

Figure 5.2: An example of a commutative reduction tree.

[1]The reduction was also used by Catanzaro, Sundaram, and Keutzer [23].

Synchronization Between Phases

The second phase can begin only after the first one, which is executed by multiple work-groups, has finished, i.e. only after values to be aggregated have been stored to the array *tmp_results* for further processing. Nevertheless, work-groups cannot synchronize between each other, as stated in Section 2.3.1, and therefore, execution of these phases has to be synchronized at the command-queue level. Hence, we defined each of these phases as standalone kernels and launched them separately.

Synchronization Between Different Levels of Reduction Tree

Launching of a separate kernel for the second phase permits different mapping of the work to work-items. Before we define it, remind ourselves that the reduction of values whose aggregation yields the result of the evaluation on a single input involves synchronization between execution of nodes on different levels. Therefore, the work on each level has to be executed using separate kernel invocation or the work on all levels has to be executed by work-items in a single work-group, since only the work-items in the same work-group may be synchronized using OpenCL C functions, as shown in Listing 5.3. We adopted the second approach, since the other one would result in assigning only the aggregation of two values to a work-item, which is a too small task.

However, synchronization between the work-items is also quite an expensive operation, and therefore, it makes sense to do as much of the reduction serially as possible. Based on this, we adjusted the way in which the reduction is computed so each work-item first computes the sum of a subcollection of the original collection serially and writes out the result into another array stored in the local memory. Only after the values stored in this array are reduced using the commutative reduction, as depicted in Listing 5.3, where the variable *size* has the value equal to the length of the array *inputs* divided by $WORK_GROUP_SIZE$. This means that when the reduction is completed the overall result is stored in this array at index 0.

```
1 __kernel void sum(constant float* inputs, int const size, global
      float* outputs)
2 {
3     int lIndex = get_local_id(0);
4     float partialSum = 0;
5     for(int i = 0; i < size; ++i) {
6         partialSum += inputs[lIndex + i * WORK_GROUP_SIZE];
7     }
8     local float partialSums[WORK_GROUP_SIZE];
9     partialSums[lIndex] = partialSum;
10    for(int offset = WORK_GROUP_SIZE / 2; offset > 0;offset >>= 1) {
11        barrier(CLK_LOCAL_MEM_FENCE);
12        if (lIndex < offset) {
13            partialSums[lIndex] +=  partialSums[lIndex + offset];
14        }
15    }
16    if (lIndex == 0) {
17      outputs[0] = partialSums[0];
18    }
19 }
```

Listing 5.3: Source code of the commutative reduction.

5.3.4 Preprocessing

In Section 5.1.1, we stated that the considered transformation functions can run in parallel without any restriction. The most fine-grained decomposition of preprocessing of multiple inputs that yields data parallel tasks conforms to assigning preprocessing of a single input to a task. In the same time, preprocessing of a single input conforms to calling maximally 30 functions, each of which takes 100 instructions in the worst case 5.1.1. This means that we would have to preprocess thousands of inputs within a single kernel invocation in order to benefit from utilization of this coprocessor. Considering these facts, we decided to execute these transformations on the host and only then move the already transformed data onto the device.

5.3.5 Data Layout

In Section 5.1 we observed that the evaluation of the model in the question on a single input conforms to data parallelism since different iterations evaluate the same function on different constants. This fact combined with the fact that the coalesced memory access has a huge performance impact, as mentioned in Section 2.5.4, determines that these constants have to be arranged in a convenient

way in order to achieve peak performance.

Each work-item accesses only one α_s^m. This implies that even if we concatenate these values to an array, then consecutive work-items in a single work-group will access this arrays at consecutive indices at the same time. This means that such arrangement of these values permits coalesced memory access.

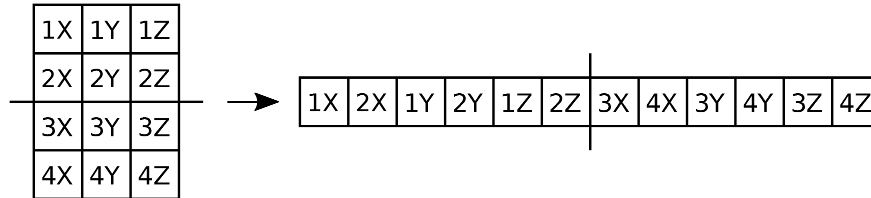

Figure 5.3: Arrangement of an array of structures/arrays as AoSoA with the small arrays sized 2.

Each work-item also accesses its own collection $\gamma_{m,s}$. Therefore, these collections have to be arranged in a much more sophisticated way, because elementary concatenation of them would result in strided memory access. To permit coalesced memory access, we have to arrange these collections either as Structure-of-Arrays (SoA) or Array-of-Structures-of-Arrays (AoSoA), whose example is depicted in Figure 5.3. We have opted for arranging these collections as AoSoA with the small arrays sized 32, since this layout provides better spatial locality then the other one.

5.3.6 Command-Queues and Kernel Synchronization

In Section 2.3.2, we stated that multiple commands may be executed at the same time only when they were submitted to an out-of-order command-queue or multiple command-queues. From these options, we decided for a single out-of-order command-queue since this option does not require initialization of a new command-queue whenever a new batch of inputs is about to be evaluated.

Due to utilizing an out-of-order command-queue we had to utilize event objects 2.3.2 to synchronize execution of dependent commands.

5.4 Programming Language, Data Formats, and Libraries

The choice of programming language is essential for any implementation. In order to select the most suitable programming language our requirements and priorities have to be specified first.

In our case, the major requirement is low evaluation latency. Due to this, we omitted all interpreted languages. The target application is also mission critical, therefore, we looked for a commonly used and well-known programming language that is robust and well maintainable. In addition, we consider explicit memory management to be quite error-prone, and therefore, we did not want to utilize languages without automatic memory management, such as C and C++.

With respect to the presented requirements, the most suitable languages were Java and C#. From these languages, we selected C# due to the fact that a company that is interested in parallelization of our AT model on the Intel Xeon Phi coprocessor has its trading model also written in C#. Consequently, an implementation written in C# will be most relevant for them.

5.4.1 Data Types

Our model approximates a function from a vector space over real numbers to real numbers and all the model constants are real numbers as well. OpenCL and .NET share 2 data types representing real numbers: single-precision and double-precision floating points. We decided for single-precision floating points, inasmuch as processing single-precision data requires fewer computation resources (RAM, cache, and bandwidth) than processing double-precision data and while still preserves a sufficient level of accuracy for this type of problems.

5.4.2 Parallel Programming in .NET

Multi-core parallelization can be achieved using both OpenCL and mechanisms provided by the selected programming language. However, we wanted to obtain a baseline that is not dependent on using OpenCL, and thus, we decided for the way of mechanisms provided by the selected language.

Three low-level mechanisms are provided by .NET to run code in parallel: *Thread*, *ThreadPool* and *Task*.

Thread represents an actual OS thread, with its own stack and kernel resources. The problem associated with using the threads is that they are costly because each of them consumes a non-trivial amount of memory for its stack. Moreover, utilization of too many threads adds CPU overhead as the processor has to perform context-switching.

ThreadPool is a wrapper around a pool of threads maintained by the Common Language Runtime (CLR)[2], whose count conforms to the number of logical cores. Using ThreadPool avoids both the overhead of creating new threads and the overhead of utilizing too many threads since it assigns tasks to threads from the

[2]The virtual machine that manages the execution of .NET programs.

pool of available threads. If all the threads are busy, tasks are enqueued until they can be served as threads become again available. On the other hand, ThreadPool provides no straightforward way of finding out when a task finished.

Task from the *Task Parallel Library* [22] combines the best of the preceding mechanisms. Tasks are assigned to threads from the thread pool by a *task scheduler*. Nevertheless, unlike ThreadPool, Task allows to find out when it finishes and to return a result.

We decided to use Task because it provides a higher level of abstraction, and therefore is easier to use and manage while its usage does not necessitate such management overhead as Thread.

Task Scheduler

A task scheduler manages how tasks are scheduled to run on thread pool threads. The default task scheduler does not provide any possibility how to specify the maximum number available threads. This means that we would not have been able to investigate how our implementation scales with this number if we had used this scheduler. Therefore, we implemented our own task scheduler (*LimitedConcurrencyLevelTaskScheduler*) that allows us to restrict the number of available threads.

5.4.3 OpenCL Host Bindings in .NET

In order to take advantage of the OpenCL framework, we need to call functions of the OpenCL API. The default API is written in C, therefore, a .NET wrapper built on the top of C API has to be utilized.

Although writing of our own wrapper should not be too complicated, we decided to use an already implemented and well-tested open-source one.

The most popular open-source wrappers are OpenCL.NET[3] and Cloo[4]. The main difference between them is that Cloo, unlike to OpenCL.NET, which is just a thin wrapper around C API, provides an object-oriented API, whose usage is illustrated in Listing 5.4. This API allows communicating with OpenCL by instance methods of objects that represent parameters passed to functions of the underlying OpenCL API. Thanks to this, Cloo provides a much more C#-like API than OpenCL.NET does. Due to this fact, we decided to use Cloo.

[3] `https://openclnet.codeplex.com/`
[4] `http://sourceforge.net/projects/cloo/`

```
// initializes buffers
var bufferMatrixA = new ComputeBuffer<float>(context,
    inputBuffersFlags, matrixA);
var bufferMatrixB = new ComputeBuffer<float>(context,
    inputBuffersFlags, matrixB);
var bufferMatrixC = new ComputeBuffer<float>(context,
    outputBufferFlags, matrixCLength);
// sets kernel arguments
kernel.SetMemoryArgument(0, bufferMatrixA);
kernel.SetMemoryArgument(1, bufferMatrixB);
kernel.SetMemoryArgument(2, bufferMatrixC);
// launches a kernel for execution
queue.Execute(kernel, null, globalWorkSize, localWorkSize, events);
// reads the result back
queue.ReadFromBuffer(bufferMatrixC, ref matrixC, true, events);
```

Listing 5.4: Source code of an application that communicates with OpenCL using Cloo.

5.5 Implementation for GPUs

In this section, we will describe how the selected approach to achieving parallelism on many-core architectures is suited for GPUs and what modifications would be required to reach their peak performance. We decided for GPUs based on the fact that they are commonly used for data-intensive computations, as stated in Section 3.2.3.

From suitability of GPUs for data-intensive computations implies that the approach to achieving parallelism utilized by our implementation is convenient even for GPUs inasmuch as it conforms to data parallelism.

Despite the previous statement, it was also stated that there are two aspects that distinguish OpenCL programming for these devices since these aspects improve the performance of OpenCL applications just on the Intel Xeon Phi coprocessors. The first of them is that the Intel Xeon Phi coprocessors do not benefit from using the local memory for caching hot data that occupy a block of the global memory. This difference is based on the fact that the Intel Xeon Phi coprocessors cache data implicitly, whereas GPUs do not. In case of our implementation, the only hot data occupying a block of the global memory that are not cached in the local memory are support vectors. That means that in order to fine-tune our implementation for GPUs we would have to add caching of these support vectors in the local memory.

The second aspect is that Intel Xeon Phi coprocessors require work-groups to take at least 100 000 clock cycles to reach the peak performance whilst GPUs

do not require such large work-groups. In case of our implementation, utilization of smaller work-groups would correspond to evaluating fewer inputs within a single block. However, evaluating fewer inputs within a block would reduce the improvement obtained by caching hot data in the local memory. Due to this, our optimization consisting in assigning the evaluation of the same support vector against a collection of inputs to a work-item is suitable even for GPUs. However, GPUs would probably require a smaller value of BS, i.e. smaller blocks in terms of included inputs than the Intel Xeon Phi coprocessor in order to reach the peak performance.

The Intel Xeon Phi coprocessor contains much more compute units than GPUs. On the other hand, compute units of GPUs contain much more processing elements than compute units of the Intel Xeon Phi coprocessor. This means that we would have to use larger work-groups in terms of contained work-items in order to efficiently utilize GPUs.

Finally, the fact that GPUs do not require such large work-groups indicates that it could be beneficial to perform the preprocessing on them. However, this hypothesis would have to be empirically validated.

6. Experimental Results

Utilization of the Intel Xeon Phi coprocessor can, on one hand, lead to higher throughput of data-intensive computations, on the other hand, it might deteriorate latency of these computations, as stated in Section 1.1. Therefore, the prototype utilizing this coprocessor, which was described in the preceding chapter, as well as the serial and multi-core prototypes, were submitted to tests in order to verify to what extent and under what conditions, e.g., the number of inputs to be evaluated, or the number of blocks executed in parallel within a single kernel invocation, utilization of this coprocessor might accelerate evaluation of our AT models.

In case of applications utilizing multiple computational units we are interested not just in the factors that interest us in case of serial applications, e.g., latency and throughput, but even in scalability with these units, as stated in Section 1.1. Therefore, the many-core and multi-core prototypes were submitted to tests in multiple configurations that differ from each other by the number of utilized units in order to examine how these prototypes scale with these units. In case of the multi-core prototype, these units correspond to threads in the thread pool 5.4.2, while in case of the many-core prototype, these units correspond to work-groups utilized within a kernel invocation. Remind ourselves that the number of work-groups utilized within a kernel invocation is determined by the number of blocks executed simultaneously within this invocation.

Constraining the number of blocks executed in parallel within a single kernel invocation does not restrict the total number of blocks that may execute in parallel since multiple kernels may execute on a device simultaneously, as stated in Section 5.3.2. Consequently, we will not be able to determine the scalability of our prototype with more computational units but rather the benefit obtained by running fewer large kernels.

6.1 Experimental Methodology

Although there are many measurable characteristics that could be investigated via testing, characteristics that interest us are solely based on the execution time. Nevertheless, even the execution time is rather a general term that may be perceived in several ways, and therefore, what it corresponds to in the remainder of this chapter has to be specified.

6.1.1 Execution Time

Applications utilized in AT act as server applications, i.e. their major responsibility is to react on occurrences of desired events. This also means that the performance of such applications should not be compared using the wall time but rather the time that they need to react to an occurrence of such event. For applications utilizing OpenCL this implies that we are not interested in the time footprint of steps that could be done only during the start-up, e.g., the context initialization and compilation of source code.

To measure the time needed to complete steps that concern us we used a real-time clock. We start to measure time right before the first input is about to be evaluated and stop right after all results are obtained.

Besides the execution time, we will also examine the time needed to modify our AT model in case of the many-core implementation. Since the modification might be required when the model has to be adapted to reflect new events occurring in the market. We will assume that the required modifications affect only the parameters of the model and not its form, i.e. the notation of its kernel method or the number of support vectors.

6.1.2 Correctness of Measured Times

Times measured by the selected method are, unfortunately, predisposed to be influenced by hardware interruptions or other processes running on the same system. Consequently, a testing methodology was proposed in order to reduce an error caused by these influences as much as possible.

Each test comprised the evaluation of one or more models on multiple inputs. Each such a test was executed $10\times$ yielding measured times $\{t_i\}_{i=1}^{10}$. When separate executions were performed in separate applications runs and within each of these runs, the test was performed twice. The objective of the first run was warming up of the hardware the test runs on, and therefore, only the execution time of the second run was measured. Using the measured times the average execution time $\bar{t}$ was computed as arithmetic average:

$$\bar{t} = \frac{1}{10}\sum_{i=1}^{10} t_i$$

Utilizing the raw average a new set T that contains only times less than or equal to $1.1 \cdot \bar{t}$ was constructed, i.e. $T = \{t_i | i \in \{1, \dots 10\} \wedge t_i \leq \bar{t}\}$. Times larger than $\bar{t}$ were omitted as they were considered to be distorted.

In the set T, we looked for a subset of cardinality 3 that includes times that differ from the raw average of this subset maximally by 1%. If there was such

a subset, we appointed the execution time t as the raw average of contained times. If multiple subsets satisfied that condition, one exhibiting the smallest variance was picked out. Finally, new measurements were executed in case that the desired subset could not be constructed.

The execution time of one model on a single input is influenced by the total number of inputs to be evaluated since the evaluation of more inputs might, for example, enable overlapping of data transfers and computations on a device and thus provides better performance. Therefore, we had to specify the minimal number of inputs after which the execution time of one model on a single input is declared to be stable. We defined this number as the minimal number of inputs yielding the execution time of one model on a single input differing maximally by 10% from times yielded by two preceding tests, i.e. tests evaluating the same number of models on fewer inputs. Numbers of inputs that were evaluated within consecutive tests formed a geometric sequence with the common ratio 2.

The measured times were not subjected to extensive statistical analysis, since we focused solely on characteristics that are computable using basic statistical methods. However, we published all triplets used to compute the execution times on the enclosed DVD, so anyone can analyze them further.

6.1.3 Hardware Specification

All tests of the serial and multi-core prototypes were performed on a computer equipped with Intel Xeon Processor E5-2630 clocked at 2.3 GHz and with 128 GB of RAM organized in 2-node NUMA[1]. Red Hat Enterprise Linux Server (version 7.1) was used as the operating system.

Tests were run using Mono runtime (version 3.12.0), which had to be patched since it malfunctioned when utilizing OpenCL. The malfunction, which caused freezing up of an application, was a consequence of the fact that Mono and OpenCL conflicted throughout using the same signal [2].

Intel Xeon Phi Coprocessor 7120P with 61 cores clocked at 1.333 GHz was used as an OpenCL device. The coprocessor was equipped with 16GB of RAM and was connected to the host via PCI-E 2.0 x16, so data could be transferred with a nominal bit rate of 8 GB/s in each direction. The above-mentioned computer was used as the host.

[1] Non-uniform memory architecture

[2] For more information about the patch see `http://stackoverflow.com/questions/17879292/cpu-killed-by-sigxcpu-using-opencl-and-mono`

6.1.4 Test Data

As we lacked access to any historical data, we performed all tests against synthetic data. Besides test data, we also generated constants used by models or used for their definition, e.g., the number of their parameters.

Test data and constants used by models were generated from a uniform distribution of an interval $\langle 0, 1)$ utilizing the *Mersenne Twister* generator, which is a pseudo-random number generator. An implementation of this generator is also contained on the enclosed DVD, so anyone can generate other testing data or even models in order to verify that our conclusions, which will be hereafter presented, are not subject to the use of appropriate values.

6.1.5 Testing Configuration

As mentioned in Section 5.3, a configuration of the many-core prototype defines the number blocks executed within a single kernel invocation. However, the number of inputs forming a single block has not been defined yet. Remind ourselves that the more inputs are evaluated within a single block the better ratio of the work assigned to a work-group to its management overhead, and withal, the more inputs are required to saturate all blocks that execute in parallel within a single kernel invocation. Considering these facts, we decided to evaluate 128 inputs within a single block. Forasmuch as this value ensures that maximally 1024 inputs are required to saturate all blocks executed within a single kernel invocation and withal the work assigned to a work-group to be complex enough.

Since our kernels do not take the number of inputs as a parameter, they always evaluate the fixed number of inputs. Therefore, the minimal number of inputs to be evaluated within a single kernel invocation equals to the number of inputs forming a single block times the number of the blocks executed simultaneously within this invocation.

Finally, note that the number of inputs evaluated within a single task in case of the multi-core prototype equals to 16.

6.2 Performance

As stated at the very beginning of this chapter, all three prototypes in various configurations were submitted to tests that consisted of the evaluation of a specified number of inputs by one or more models in order to compare the performance, which is evaluated only in the execution time, of diverse configurations of these prototypes depending on the number of models to be evaluated and the number of inputs.

6.2.1 Measured Times

Following tables contain measured times along with optimal speed up, which is derived from the stable times, to the serial prototype for 1, 64, and 128 models. In those tables, columns titled with s contain values for the serial prototype, columns titled with t_i contain values for the multi-core prototype utilizing maximally i threads and columns b_i contain values for the many-core prototype executing i blocks simultaneously, which conforms to utilizing $32 \cdot i$ work-groups. We have to highlight that emphasized values represent times derived from the stable times.

Tables containing values for 2, 4, 8, 16, and 32 models will not be presented here since these tables would contain fairly similar values referenced to the number of models times the number of inputs as tables for 1 or 64 models. Likewise, times for the multi-core prototype utilizing maximally 2 threads will not be presented, since it turned out that the multi-core prototype scales nearly linearly up to 4 threads, and hence, these times would not have provided us any valuable information.

Testing a configuration on a number of inputs that does not saturate configurations utilizing fewer computational units would not have provided us any meaningful information. Therefore, we did not start testing any configuration on the minimal number of inputs defined above but rather on the number of inputs that saturate the closest previous configuration in terms of the number of utilized computational units.

All three presented tables indicate that for the corresponding numbers of models our prototype in any configuration performs better than the multi-core prototype utilizing up to 8 threads. Moreover, when we consider only configurations executing 2 and more blocks in parallel then our prototype performs even better than the multi-core prototype utilizing 16 threads. The way in which the multi-core prototype scales foreshadows that our prototype utilizing 8 blocks would outperform even the multi-core prototype utilizing 32 threads.

The presented tables also indicate that a configuration of our prototype is saturated by one-quarter of the work that saturates a configuration of the multi-core prototype that provides the roughly same speed up as the considered configuration of our prototype.

Inputs	s	t_4	t_8	t_{16}	b_1	b_2	b_4	b_8
8	0.004							
16	0.007							
32	0.014							
64	0.029							
128	*0.057*				0.007			
256	*0.115*				0.011	0.009		
512	*0.230*	0.230			0.022	0.014	0.013	
1024	*0.459*	0.238			0.043	0.028	0.020	0.019
2048	*0.919*	0.403			*0.085*	0.057	0.041	0.037
4096	*1.837*	0.560	0.560		*0.171*	0.112	0.080	0.073
8192	*3.675*	0.920	0.818		*0.341*	*0.223*	0.149	0.143
16384	*7.349*	1.892	1.309	1.307	*0.683*	*0.446*	0.293	0.250
32768	*14.698*	3.783	1.911	1.669	*1.365*	*0.893*	0.582	0.481
65536	*29.397*	*7.565*	3.826	2.415	*2.731*	*1.786*	*1.165*	0.936
131072	*58.794*	*15.131*	7.730	4.205	*5.461*	*3.571*	*2.329*	*1.872*
262144	*117.588*	*30.262*	*15.461*	8.403	*10.923*	*7.142*	*4.659*	*3.744*
524288	*235.176*	*60.524*	*30.921*	16.771	*21.846*	*14.284*	*9.318*	*7.488*
Speedup	1×	3.89×	7.61×	14.02×	10.77×	16.46×	25.24×	31.41×

Table 6.1: Evaluation times for 1 model in seconds.

Inputs	s	t_4	t_8	t_{16}	b_1	b_2	b_4	b_8
8	0.238							
16	0.476	0.240						
32	0.954	0.359						
64	1.905	0.537						
128	3.812	0.957			0.356			
256	*7.624*	1.910	1.151		0.735	0.472		
512	*15.248*	3.920	1.983		1.474	0.946	0.581	
1024	*30.496*	*7.841*	3.962	2.457	*2.948*	1.812	1.117	0.946
2048	*60.993*	*15.681*	7.921	4.367	*5.896*	*3.625*	2.267	1.922
4096	*121.985*	*31.363*	*15.843*	8.705	*11.791*	*7.249*	*4.535*	3.721
8192	*243.971*	*62.725*	*31.686*	17.402	*23.582*	*14.498*	*9.069*	*7.441*
Speedup	1×	3.89×	7.7×	14.02×	10.35×	16.83×	26.9×	32.79×

Table 6.2: Evaluation times for 64 models in seconds.

Inputs	s	t_4	t_8	t_{16}	b_1	b_2	b_4	b_8
8	0.480	0.222						
16	0.964	0.322						
32	1.921	0.511						
64	3.845	0.961						
128	7.688	1.922	1.139		0.848			
256	*15.376*	3.841	1.987		1.649	1.058		
512	*30.752*	*7.682*	3.974	2.595	3.373	2.158	1.320	
1024	*61.505*	*15.363*	7.971	4.376	*6.747*	4.382	2.612	1.898
2048	*123.010*	*30.726*	*15.942*	8.731	*13.493*	*8.764*	5.193	3.749
4096	*246.019*	*61.452*	*31.883*	17.483	*26.986*	*17.527*	*10.387*	7.622
Speedup	1×	4×	7.72×	14.07×	9.12×	14.04×	23.69×	32.28×

Table 6.3: Evaluation times for 128 models in seconds.

The last table provides slightly different values referenced to the number of inputs evaluated multiplied by the number of models than the other presented tables. Along with this, we had repeated the whole tests for the corresponding number of models many times, since measured times often did not fulfil conditions determined by our testing methodology. Both these facts might indicate restricted usability of our prototype in terms of the number of models to evaluate. Therefore, our prototype was submitted to tests that consist of the evaluation of 256 models in order to validate this hypothesis.

The measured times for this number of models validate this hypothesis inasmuch as obtained stable times were equal approximately to 1.3× of the stable times for 128 models. Besides, even the tests had to be executed more times than in case of 128 models.

Restricted usability of our prototype is most likely caused by the fact that data accessed within the evaluation of models, i.e. the constants of these models, and transformed inputs, do not fit into cache when a large number of models should be evaluated. It is also possible that even overhead of scheduling grows with the number of models to evaluate.

6.2.2 Scalability

Now, we will take a more thorough look at measured times for our prototype from the perspective of scalability with the number of blocks executed within a single kernel invocation.

To determine this feature of our prototype we will consider only the stable times, which were defined in Section 6.1.2. Due to this, results that will be further

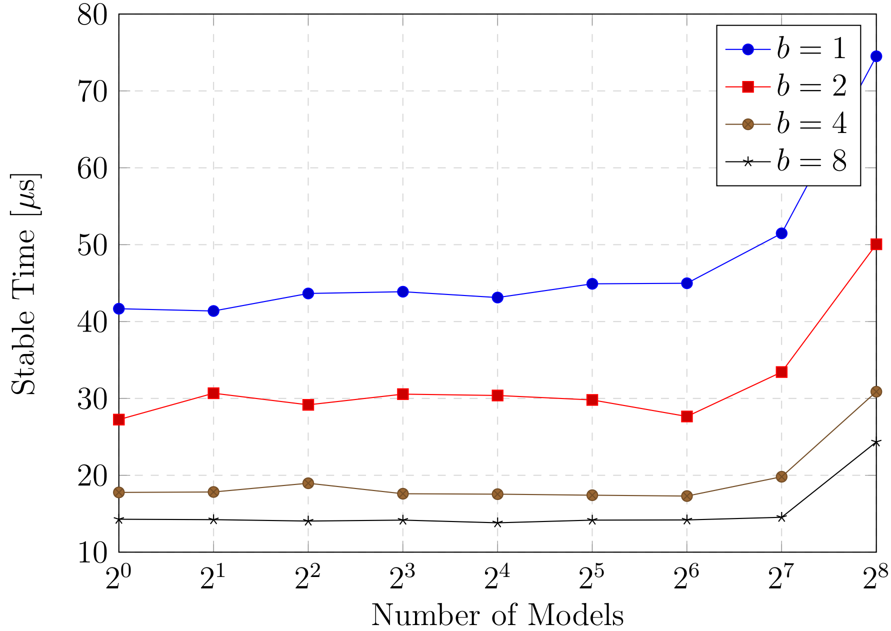

Figure 6.1: Stable times of different configurations.

presented are valid only when a sufficient number of inputs is evaluated, since different configurations reach their stable times for different numbers of inputs.

From Figure 6.1, where the value of b denotes the number of blocks executed in parallel within a single kernel invocation, we can deduce that the more blocks are executed in parallel the better performance is provided. In other words, launching fewer kernels evaluating larger batches of inputs is more efficient than the opposite approach of launching many kernels processing smaller batches of inputs. In fact, this ascertainment is fully compliant with the recommendation stated in Section 3.2.1 to execute as many work-groups in a single kernel invocation as possible. On the other hand, it shows that employing all logical cores at once is not sufficient to reach the peak performance since multiple invocations of a configuration that utilizes only a subset of all logical cores could employ all logical cores at once as they are executed out-of-order.

Factors contributing to better performance of configurations executing more blocks in parallel are necessarily a smaller ratio of the management of the kernel execution to the actually executed work and withal a larger data transfer throughput. Table 6.4, which includes times required to transfer data representing a relevant number of inputs along with speed up referenced to the time of transferring 128 inputs, i.e. the smallest batch of inputs executed within a single kernel invocation is presented in order to illustrate the impact of transferring data in larger batches.

Inputs	128	256	512	1024
Time	589.2	592.77	641.07	705.83
Speedup	1×	1.99×	3.67×	6.67×

Table 6.4: Transfer times for relevant numbers of inputs in microseconds.

6.3 Cost of Modifications

In this section, we will examine how costly in terms of time the modifications of our model are. Note that the considered modifications alter only parameters of our model, e.g., support vectors, or their weights, as stated in Section 6.1.1.

Parameters to be modified are immutable within multiple kernel invocations, and hence, it is favorable to locate these parameters to the constants memory, since this memory is better cached than the global memory 2.4. To place parameters into the constant memory they have to be:

- defined as compile time constants,
- received as parameters specified with the *constant* address qualifier.

Defining parameters of our model as compile time constants forces the kernel source code to be modified and rebuilt in order to modify our model. Receiving these parameters as parameters of a kernel invocation requires initializing new buffers and setting these buffers to appropriate kernel arguments. Times required to modify our model using both these approaches are provided in Table 6.5.

Approach	Recompilation	Switching buffers
Time	707.16	2

Table 6.5: Times required to modify parameters of our model in milliseconds.

The above-referenced table indicates that the second approach that modifies our model using new buffers allows much faster modification than the other one.

On the other hand, both these implementations may not provide the same performance. In order to compare their performance, we decided to submit even the implementation of the second approach in a configuration executing 8 blocks in parallel to tests. These tests conformed to evaluating 1 to 64 models on numbers of inputs that were used to define the stable times for these numbers of models in the preceding section.

From Figure 6.2, which presents measured times for both implementations, ensues that the implementation utilizing hard-coded constants outperforms the second implementation when more than two models are evaluated, and even more, that difference in performance between these implementation increases with the

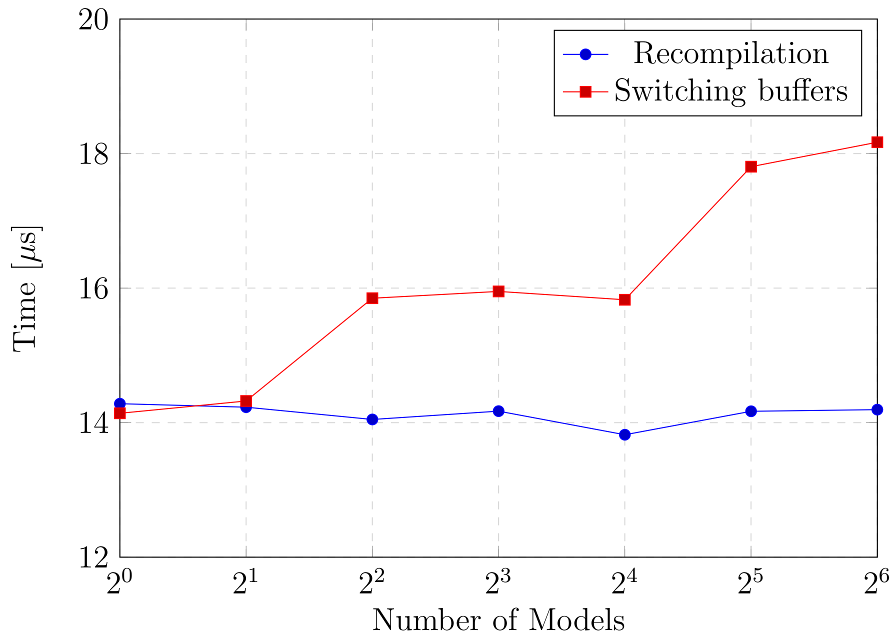

Figure 6.2: Evaluation times of a single model on one input for different approaches.

number of models. This phenomenon is likely caused by the fact that accessing hard-coded constants is more easily optimized than accessing parameters residing in constant memory. Realize that this assumption is compliant with the fact that this phenomenon occurred only when multiple models were evaluated, i.e. when accessing to parameters of multiple models should have been optimized.

7. Conclusion

The main objective of the presented thesis was to analyze existing models for algorithmic trading from the perspective of parallel programming and based on this analysis design and implement a prototype intensively utilizing a highly parallel architecture, namely the Intel Xeon Phi coprocessor.

Before we started dealing with the analysis of the AT models in question, we had described the form of these models and briefly introduced the theory behind them. This introduction was primarily focused on highlighting advantages of our AT models in comparison with other commonly used AT models.

Thereafter, we proposed several approaches to the parallel evaluation of one or more our AT models on one or more inputs. For each of these approaches, we discussed its suitability for the selected coprocessor considering provided facts about OpenCL programming for highly parallel architectures, the Intel Xeon Phi architecture, and specifics of OpenCL programming for this coprocessor.

The best-suited approach was implemented in our prototype, which was submitted to tests just as a serial and multi-core baseline. The results indicate that the proposed solution would perform roughly as the multi-core prototype utilizing 32 threads, and moreover, it would require approximately one-quarter of the work compared to the multi-core prototype to reach the peak performance. The results also indicate that utilization of this solution would be beneficial even when evaluating just tens of inputs, which means that the overhead introduced by offloading the evaluation to the coprocessor is negligible compared to the performance gain.

The results furthermore indicate that performance of the proposed solution decreases when more than 64 models are evaluated at once. We stated factors that most likely cause this phenomenon, nevertheless, further research on this phenomenon would be required when such a number of models should be commonly evaluated.

Even though there are a few questions concerning the usability of our prototype, we have successfully achieved our objectives and proved that the evaluation of the AT models based on multiple kernel support vector regression is a highly parallelizable problem.

7.1 Future Work

Since our implementation partitions the evaluation of one model on one input to two phases, the first of them is being perfectly parallelizable while the second phase achieves parallelism using a well-established approach, it is quite likely that

there are no options for improving this implementation in terms of parallelization. Hence, possible improvements would lay in fine-tuning parameters of our implementation to maximize its performance when evaluating the expected number of models on the expected number of inputs. In other words, improvements would correspond to fine-tuning the number of inputs evaluated within a single block and the number of blocks executed in parallel within a single kernel invocation to reach the peak performance under common conditions.

It should be also investigated how beneficial could it be to utilize other OpenCL devices, primarily GPUs. Note that we proposed modifications required to adapt our implementation for GPUs in Section 5.5. When investigating performance of the adjusted implementation, it should be also examined whether it would be beneficial to perform preprocessing on GPUs or not, as suggested in the above-referenced section.

Bibliography

[1] *Portfolio Selection on JSTOR.* [online] [Accessed: 6. 5. 2016]
`http://www.jstor.org/stable/2975974`

[2] *The fast and furious | The Economist.* The Economist. [online] 2. 2012 [Accessed: 6. 5. 2016]
`http://www.economist.com/node/21547988`

[3] T. G. Mattson, B. A. Sanders, B .A. Massingill. *Patterns for Parallel Programming.* 1st edition. Boston: Addison-Wesley Professional, 2004. ISBN 0321940784

[4] *The OpenCL Specification Version 1.2.* [online] [Accessed: 6. 5. 2016]
`https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`

[5] *The OpenCL C Specification Version 2.0.* [online] [Accessed: 6. 5. 2016]
`https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf`

[6] A. Munshi, B. Gaster, T. G.. Mattson, J. Fung, D. Ginsburg. *OpenCL Programming Guide.* 1st edition. Boston: Addison-Wesley Professional , 2011. ISBN 0321749642

[7] R. Rezaur. *Intel Xeon Phi Coprocessor Architecture and Tools, The Guide for Application Developers* New York: Apress, 2013. ISBN 978-1-4302-5926-8

[8] Ch. Demerjian. *Intel details Knights Corner architecture at long last - SemiAccurate* [online] 9. 2012 [Accessed: 6. 5. 2016]
`http://semiaccurate.com/2012/08/28/intel-details-knights-corner-architecture-at-long-last/`

[9] S. Li. *Memory Management for Optimal Performance on Intel®Xeon Phi™Coprocessor: Alignment and Prefetching* [online] 3. 2014 [Accessed: 6. 5. 2016]
`https://software.intel.com/en-us/articles/memory-management-for-optimal-performance-on-intel-xeon-phi-coprocessor-alignment-and`

[10] *OpenCL* Design and Programming Guide for the Intel®Xeon Phi™Coprocessor — Intel®Software* 2. 2014 [Accessed: 7. 7. 2016]
`https://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor`

[11] M. S. Papamarcos,J. H. Patel. *A low-overhead coherence solution for multiprocessors with private cache memories.* ACM SIGARCH Computer Architecture News 12(3), 1998. 284-290

[12] B. McClure. *Fundamental Analysis: What Is It? — Investopedia* [online] [Accessed: 6. 5. 2016]
`http://www.investopedia.com/university/fundamentalanalysis/fundanalysis1.asp`

[13] S. A .M. Yaser, A. F. Atiya. *Introduction to financial forecasting.* Applied Intelligence, 1996. 6: 205–213

[14] L. Cao, F. E. H. Tay. *Financial Forecasting Using Support Vector Machines.* Neural Comput & Applic 2001. 10: 184–192

[15] H.Tong, K. S. Lim. *Threshold Autoregression, Limit Cycles and Cyclical Data.* Journal of the Royal Statistical Society, Series B (Methodological), 1980. 42.3: 245–292

[16] R. F. Engle. *Autoregressive conditional heteroskedasticity with estimates of the variance of UK inflation.* Econometrica 50, 1982. 987–1008

[17] C. Sammut, G. Webb. *Encyclopedia of Machine Learning.* Boston: Springer US, 2010. ISBN 978-0-387-30164-8

[18] V. N. Vapnik. *The Nature of Statistical Learning Theory.* New York, Springer-Verlag. 1995

[19] M. Gonen, E. Alpaydin. *Multiple Kernel Learning Algorithms.* Journal of Machine Learning Research 12, 2011. 2211-2268

[20] V. N. Vapnik, S. E. Golowich, A. J. Smola. *Support vector method for function approximation, regression estimation, and signal processing.* Advances in Neural Information Processing Systems, 1996. 9: 281–287

[21] I. Landwerth. *The JIT finally proposed. JIT and SIMD are getting married. — .NET blog* [online] 5. 2014 [Accessed: 7. 7. 2016]
`https://blogs.msdn.microsoft.com/dotnet/2014/04/07/the-jit-finally-proposed-jit-and-simd-are-getting-married/`

[22] G. C. Hillar. *Professional Parallel Programming with C#: Master Parallel Extensions with .NET 4*. Birmingham: Wrox, 2010. ISBN 978-0-470-49599-5

[23] B. Catanzaro, N. Sundaram, K .Keutzer. *Fast Support Vector Machine Training and Classification on Graphics Processors.* International conference on Machine learning, 2008. 104-111

[24] S. Herrero-Lopez, J. R. Williams, A. Sanchez. *Parallel Multiclass Classification using SVMs on GPUs.* Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010. 2-11

List of Figures

2.1 A schema of the OpenCL platform model. 9
2.2 An example of how the global IDs, the local IDs, and work-group IDs are related. The shaded block has the global ID of (g_x,g_y) = (6,5) and the local ID of (l_x,l_y) = (2,1). 10
2.3 The mapping of the OpenCL memory model to the OpenCL platform model. 16
2.4 An example of transfers and executions pipelining. 17

3.1 The Intel Xeon Phi processor microarchitecture [8] 20
3.2 An example of conditional updating (x represents an unchanged value). 22
3.3 The memory architecture of the Intel Xeon Phi coprocessor. . . . 23

4.1 An example of overfitting. 31
4.2 An instance of SVM. 32
4.3 An example of mapping of non-separable inputs to a feature space. 36

5.1 Mapping of inputs to blocks in case of 4 blocks where each of these blocks processes 2 inputs. 48
5.2 An example of a commutative reduction tree. 49
5.3 Arrangement of an array of structures/arrays as AoSoA with the small arrays sized 2. 52

6.1 Stable times of different configurations. 64
6.2 Evaluation times of a single model on one input for different approaches. 66

List of Tables

6.1 Evaluation times for 1 model in seconds. 62
6.2 Evaluation times for 64 models in seconds. 62
6.3 Evaluation times for 128 models in seconds. 63
6.4 Transfer times for relevant numbers of inputs in microseconds. . . 65
6.5 Times required to modify parameters of our model in milliseconds. 65

List of Abbreviations

AoS Array-of-Structures. 19, 47

AoSoA Array-of-Structures-of-Arrays. 19, 52, 72

ARIMA AutoRegressive Integrated Moving Average. 29, 30

AT Algorithmic Trading. 4, 40, 57, 58, 67

CLR Common Language Runtime. 53

EMA Exponential Moving Average. 29

HFT High-Frequency Trading. 4, 28

MIC Many Integrated Core. 6, 20

MKL Multiple Kernel Learning. 36

SIMD Single Instruction, Multiple Data. 9

SMA Simple Moving Average. 29

SoA Structure-of-Arrays. 19, 52

SVM Support Vector Machine. 32–35, 37, 72

SVR Support Vector Regression. 37

TD Tag Directory. 23

VPU Vector Processing Unit. 20, 21, 24, 25, 46

WMA Weighted Moving Average. 29

Attachments

The directory on the enclosed DVD has following structure:

- */doc*
  - */thesis* - contains this document in PDF and PostScript format and the corresponding source files
  - */results* - provides all triplets of measured times that were used to compute the execution time for every test
  - */Documentation.chm* - reference documentation of code generated by Sandcastle Help File Builder[1] from source code
- */src* - contains source code of our implementation, the serial and multi-core prototype, the Mersenne Twister generator, and the Cloo library with project files for Microsoft Visual Studio 2015
- */Mono.3.12.0.tar.gz* - packed source code of the patched Mono (3.12.0)
- */ReadMe.txt* - provides instructions on installing Mono from source code on Linux and instructions on repeating/performing other tests

[1] `https://shfb.codeplex.com/`