Speeding Up Value at Risk Calculations Using Accelerators

JONAS AF SANDEBERG
Speeding Up Value at Risk Calculations Using Accelerators

A performance comparison of Value at Risk calculations on different types of processing units

JONAS AF SANDEBERG

Degree project in II21X at KTH Information and Communication Technology
Supervisors: Tomas Forsberg, Peter Snellman
Examiner: Mats Brorsson
Abstract

Calculating Value at Risk (VaR) can be a time consuming task. Therefore it is of interest to find a way to parallelize this calculation to increase performance. Based on a system built in Java, which hardware is best suited for these calculations? This thesis aims to find which kind of processing unit that gives optimal performance when calculating scenario based VaR. First the differences of the CPU, GPU and coprocessor is examined in a theoretical study. Then multiple versions of a parallel VaR algorithm are implemented for a CPU, a GPU and a coprocessor trying to make use of the findings from the theoretical study. The performance and ease of programming for each version is evaluated and analyzed. By running performance tests it is found that the CPU was the winner when coming to performance while running the chosen VaR algorithm and problem sizes.
Referat

Uppsnabbning av riskberäkningar

Contents

List of Figures viii
List of Tables x

1 Introduction 1
  1.1 Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.2 Value at Risk . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.3 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
    1.3.1 Theoretical study . . . . . . . . . . . . . . . . . . . . . . . . . 2
    1.3.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 2
    1.3.3 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
    1.3.4 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
  1.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

I Background 5

2 Value at Risk 7
  2.1 Scenario based VaR . . . . . . . . . . . . . . . . . . . . . . . . . . 7
    2.1.1 Historical VaR . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  2.2 Parametric VaR . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  2.3 Monte Carlo Simulation . . . . . . . . . . . . . . . . . . . . . . . . . 9
    2.3.1 Simulating with one random variable . . . . . . . . . . . . . . 9

3 Processing Units 11
  3.1 Multicore CPU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
    3.1.1 Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
    3.1.2 Programming model . . . . . . . . . . . . . . . . . . . . . . . 12
    3.1.3 Strengths and weaknesses . . . . . . . . . . . . . . . . . . . . 14
  3.2 GPU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
    3.2.1 NVIDIA Kepler Architecture . . . . . . . . . . . . . . . . . . . 14
    3.2.2 Memory architecture . . . . . . . . . . . . . . . . . . . . . . . 17
    3.2.3 Programming model . . . . . . . . . . . . . . . . . . . . . . . . 17
    3.2.4 Strengths and weaknesses . . . . . . . . . . . . . . . . . . . . 18
CONTENTS

3.2.5 Programming Languages .................................................. 18
3.2.6 OpenCL ................................................................. 19
3.2.7 Aparapi ................................................................. 21
3.3 Coprocessors ................................................................. 22
  3.3.1 Intel Xeon Phi Architecture ........................................... 22
  3.3.2 Strengths and weaknesses ............................................. 24
3.4 Vector instructions ............................................................ 24
  3.4.1 Strengths and weaknesses ............................................. 24

II Results 25

4 Implementation 27
  4.1 VaR Algorithm ............................................................ 27
  4.2 Properties of the problem ................................................. 28
    4.2.1 Parallelizing the calculation ..................................... 28
  4.3 Serial reference version ................................................ 29
  4.4 GPU ................................................................. 30
    4.4.1 Different versions .................................................. 30
    4.4.2 Local memory ........................................................ 33
    4.4.3 Vectors ............................................................. 34
    4.4.4 Language bindings .................................................. 36
    4.4.5 Aparapi ............................................................. 36
  4.5 Coprocessor ............................................................... 37
    4.5.1 OpenCL ............................................................. 37
  4.6 Vector instructions ........................................................ 37
  4.7 Java8 ................................................................. 38
  4.8 Implementations ........................................................... 39
  4.9 Expectations .............................................................. 40
    4.9.1 Expected winner .................................................... 41

5 Performance tests method 43
  5.1 Hardware ................................................................. 43
  5.2 What to measure .......................................................... 43
  5.3 Performance tests in Java ................................................ 44
  5.4 VaR performance tests ................................................... 44
  5.5 Optimizing each version .................................................. 45

6 Evaluation 49
  6.1 Ease of programming ....................................................... 49
    6.1.1 OpenCL vs Aparapi ................................................. 49
    6.1.2 OpenCL: GPU vs CPU vs Xeon Phi ................................ 50
    6.1.3 Java8 ............................................................... 50
  6.2 Correctness ............................................................... 50
CONTENTS

6.2.1 Accuracy ................................................. 51
6.3 Performance results ...................................... 52
  6.3.1 Performance with doubles ............................ 52
  6.3.2 Performance with floats ............................. 52
6.4 Analysis .................................................... 56
  6.4.1 Winner of the performance tests ...................... 56
  6.4.2 Performance of Aparapi ............................... 56
  6.4.3 The optimal GPU version ............................. 57
  6.4.4 Overhead on GPU and coprocessor ................... 57
  6.4.5 Economical aspect .................................. 59
6.5 The verdict ................................................. 60
6.6 Future research ........................................... 60

III Bibliography ............................................... 61

Bibliography ................................................... 63

Appendices ..................................................... 67

Glossary ........................................................ 69
  Definitions .................................................. 69
  Abbreviations .............................................. 69

Test Data ....................................................... 71
List of Figures

3.1 Lambda examples .............................................. 12
3.2 Vector addition with parallel streams ...................... 13
3.3 Fork/Join pseudo code ........................................ 14
3.4 Streaming multiprocessor in NVIDIA Kepler [1] .......... 15
3.5 Overview of the Kepler Architecture [1] ...................... 16
3.6 The Kepler memory hierarchy[1] .............................. 17
3.7 Kernel for vector addition in OpenCL ....................... 19
3.8 Starting an OpenCl kernel in Jogamp’s JOCL ............... 20
3.9 Vector addition in Aparapi ................................. 21
3.11 Architecture of a core in the Xeon Phi [2] ............... 23

4.1 One way to split the multiplication of a row. Each cell represents a single element ................................. 29
4.2 Tiled serial implementation ................................... 30
4.3 OpenCL kernel where each thread calculates a whole row .................................................. 31
4.4 Three different ways to partition the calculation ........... 32
4.5 OpenCL kernel for summarizing partial results from partitioned rows ............................................. 32
4.6 OpenCL kernel where each thread calculates a partitioned row as in Figure 4.4b ...................................... 33
4.7 OpenCL kernel where each thread calculates a full row and utilizes local memory .................................. 34
4.8 OpenCL kernel where each thread calculates a partitioned row as in Figure 4.4c and utilizing the vector datatype .................................................. 35
4.9 Aparapi kernel where each thread calculates a full row .................................................. 36
4.10 How to hint the JIT compiler about vectorization .......... 38
4.11 Matrix vector multiplication in Java8 ........................ 38

5.1 Java8 Speedup per thread on Bacall .......................... 47

6.1 Speedup Valarauko 256 x 1024 ............................... 53
6.2 Speedup Valarauko 1024 x 1024 ............................... 53
6.3 Speedup Valarauko 256 x 39936 ............................... 54
6.4 Speedup Valarauko 1024 x 39936 ............................... 54
List of Figures

6.5 Speedup on Sammy ............................................. 55
6.6 Speedup on Bacall ............................................. 55

1 Runtimes Valarauko 256 x 1024 ................................. 71
2 Runtimes Valarauko 1024 x 1024 ................................. 72
3 Runtimes Valarauko 256 x 39936 ................................. 72
4 Runtimes Valarauko 1024 x 39936 ................................. 73
5 Runtimes Bacall .................................................. 73
6 Runtimes Sammy 1024 risk factors ............................... 74
7 Runtimes Sammy 39936 risk factors ............................... 74
8 Runtimes per thread on Bacall 256 x 1024 ...................... 75
9 Runtimes per thread on Bacall 1024 x 1024 ...................... 75
10 Runtimes per thread on Bacall 256 x 39936 ................... 76
11 Runtimes per thread on Bacall 1024 x 39936 ................... 76
# List of Tables

4.1 Scenario matrix with 3 scenarios and 2 risk factors . . . . . . . . . . . . 27  
4.2 Portfolio with 2 exposures . . . . . . . . . . . . . . . . . . . . . . . . . 28  
4.3 Matrix-vector multiplication . . . . . . . . . . . . . . . . . . . . . . . . 28  
4.4 Resulting Profit & Loss vector . . . . . . . . . . . . . . . . . . . . . . 28  
5.1 What to measure for a fair performance comparison . . . . . . . . . . . . 44  
5.2 Matrix sizes used in the performance tests . . . . . . . . . . . . . . . . . 45  
5.3 Best serial and parallel Java version. Specifying the number of rows and  
    columns in the two cases where the blocked algorithm performed best . 46  
5.4 Partitions per row . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46  
5.5 Best workgroup sizes . . . . . . . . . . . . . . . . . . . . . . . . . . . 47  
6.1 Number of correct decimals with doubles . . . . . . . . . . . . . . . 51  
6.2 Number of correct decimals with floats . . . . . . . . . . . . . . . . 51  
6.3 Overhead for transfer of data and the theoretical execution time without  
    the overhead for the GPU on Valarauko . . . . . . . . . . . . . . . . . . 58  
6.4 Overhead of JNI call and the theoretical execution time without the  
    overhead for the GPU on Valarauko . . . . . . . . . . . . . . . . . . . . 58  
6.5 Total overhead and the theoretical execution time without the overhead  
    for the GPU on Valarauko . . . . . . . . . . . . . . . . . . . . . . . . . 59  
1  Glossary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69  
2  Abbreviations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
Chapter 1

Introduction

Cinnober markets a system for real-time clearing of portfolios of financial instruments. An important part of a clearing business is the risk management. The risk models are used both to manage the margin levels required by clearing members to cover their risks, and to manage the process by which the clearing house handles a member default event, should such an event take place.

The calculation of the margin requirement can be time consuming. Therefore it is of interest to find ways to speed up this calculation, especially if it is performed in real-time.

1.1 Goal

The goal for the thesis work is to evaluate if and how general-purpose computing on graphics processing units (GPGPU), or other types of coprocessors, can be applied to risk computation in a system written in Java.

The goal is a performance evaluation of the selected Value at Risk algorithm on different processing units.

1.2 Value at Risk

Value at Risk (VaR) is a risk measure that is used to measure the estimated maximum loss of a financial portfolio given a particular confidence level and time horizon [3]. An example could be a confidence level of 99% and a time frame of one day. Lets say this gives you a VaR of $1000. This means that with 99% probability you will lose at most $1000 in a day. Thus it does not give you the actual worst possible loss but an estimated worst possible loss with a small risk of it being even greater.

This measure can be calculated in a lot of different ways [3] but it is normally a time consuming calculation for a mixed portfolio, i.e. a portfolio containing positions in multiple instruments.
1.3 Approach

This thesis consists of a theoretical study, implementations of the algorithm and an evaluation of the implementations.

1.3.1 Theoretical study

The following has been studied prior to the evaluation:

- A background of different approaches to parallel computing e.g. GPGPU, coprocessors and vector instructions.
- A background on the programming model for GPGPU programming.
- The basics of Value-at-Risk algorithms.
- A short study on the available Java language bindings that can be used to implement GPU kernels from Java.
- A short study on the available and relevant GPU frameworks, like Aparapi.

1.3.2 Implementation

The VaR algorithm is implemented on a GPU, a coprocessor and on a CPU with vector instructions. Implementations have been done both in OpenCL (see Section 3.2.6) and in Java with the high level framework Aparapi (see Section 3.2.7). Two versions in pure Java has also been implemented. One serial version that uses vector instructions (see Section 3.4) and one parallel using parallel streams in Java8 (see Section 4.7). Some different optimizations have also been considered to improve performance of the different implementations. To be able to evaluate the correctness of the implementations two serial versions have also been implemented. These are also used to calculate the speedup gained from the parallelization.

The following versions were implemented:

- GPU
  - OpenCL
  - Aparapi
- Coprocessor
  - OpenCL
- CPU
  - OpenCL
  - Aparapi
  - Java with vector instructions
  - Java8 with parallel streams
  - Serial reference implementation in Java
1.4. RESULTS

1.3.3 Evaluation

The following evaluations have been done:

- The correctness of the algorithm by comparing the results with a reference algorithm.
- The performance of the algorithm running on a GPU, coprocessor and CPU with vector instructions.
- Differences in performance and ease of programming in OpenCL versus the framework Aparapi.
- Difference in accuracy and performance using single-precision versus double-precision floating-points.

1.3.4 Limitations

The following limitations have been applied to the thesis:

- The GPU implementations are done in OpenCL. This way it is possible to use the same language for implementations on multiple brands of GPUs as well as coprocessors.
- The systems at Cinnober are developed in Java. Therefore language bindings to use GPU kernels from within Java have been used to make a possible future integration easy.

1.4 Results

From the theoretical study the strengths and weaknesses of the different processing units are revealed. A multicore CPU is the obvious choice when aiming for high speed, while still providing the ability of parallel computations. The coprocessor and GPU instead aim at as high throughput as possible rather than speed. As scenario based VaR algorithms are data parallel they could potentially perform well on a coprocessor or GPU if the problem size is large enough.

The performance tests performed in the thesis show that parallelizing scenario based VaR calculations gives a good speedup compared to a serial implementation. It also shows that it does not appear to be worth the trouble of using a coprocessor or GPU as the parallel version written only in Java gave best performance, even after multiple optimizations to the GPU implementation. The high level framework Aparapi appears to have some way to go before reaching the same performance level that can be achieved by writing your own OpenCL kernels.

Two disadvantages of running kernels on the GPU and coprocessor were examined to see if they had cost the coprocessor and GPU the top position in the tests. These devices are connected via PCI Express. This means that the data for the computation has to be copied to the device before the calculation can start.

The second disadvantage comes from that the systems at Cinnober are written in Java. Therefore the kernels have to be called from within a Java JVM. Both the
extra time from the copy and the JNI call added together still does not change the performance results, Java is still faster in the performed tests.

Looking from an economical aspect the pure Java implementation is also favorable. It has a higher performance per watt rating than the other processing units which will make it cheaper to keep the system up and running. When writing large programs the cost of the implementation is likely a lot higher than the cost of the hardware. For companies like Cinnober where Java is the main language for most employees the implementation time will be shorter when writing Java and thus the cost will be lower too.
Part I

Background
Chapter 2

Value at Risk

As stated in Section 1.2 Value at Risk (VaR) is a risk measure that is used to measure the estimated maximum loss of a financial portfolio given a particular confidence level and time horizon. At Cinnober VaR is used for both margin requirement calculation and portfolio risk analysis.

In this chapter some of the different ways to calculate VaR are described.

2.1 Scenario based VaR

One of the more straightforward ways to calculate VaR is with a scenario based calculation. In the scenario based calculations you specify risk factors that could affect the value of your portfolio and set up different scenarios detailing how each of these factors change. Risk factors can be almost anything, though a simple example is the rate of a specific currency pair or the price of some underlying asset. The scenarios are then to reflect different changes to these factors. One scenario could be that the exchange rate of a particular currency pair goes up while the value of the asset goes down. Another scenario could be that both the exchange rate and the value of the asset goes up and so on.

2.1.1 Historical VaR

Historical VaR or the historical simulation method is a special case of scenario based VaR. In the historical simulation method you look at the historical data for the risk factors [3]. Imagine the price of gold as an example. By looking at how the price has changed in the past, it is possible to calculate the VaR with the desired confidence level. The VaR with a 99% confidence level is the 99th percentile of the collected returns. If the returns were collected daily the time horizon would be one day. It is possible to simulate multi-day horizons from one-day historical data by calculating the change over two or more days instead of one.

Historical VaR is easy to calculate and does not need any assumptions about the distribution of prices except implicitly if we apply volatility normalization. When
CHAPTER 2. VALUE AT RISK

having volatility normalization, assumptions to estimate the daily volatility are needed in the historical dataset. One big advantage (and a fundamental assumption) is that the correlation structure, if any, is assumed to be reflected in the historical outcomes of the risk factors. It can come with some disadvantages though, such as if all the historical prices have the same weight. By applying different weight according to some model it is possible to prevent this. Another disadvantage is that it assumes that the immediate future is fairly represented by the past, though depending on the look-back period some important events might not be included. A 99% confidence level and daily horizon with a historical data of 100 days will in average contain only one value in the tail of 1%. To get a meaningful VaR measure it is therefore necessary to gather historical data for a long period. This in turn might lead to the use of old data that is no longer relevant. A tradeoff has to be made between higher precision from a large data set and the possibility of misdirection from old data.

2.2 Parametric VaR

If it is reasonable to estimate the risk factor outcomes as a parametric distribution, e.g. normal distribution, it is possible to use the parametric approach to VaR [3]. This can simplify the calculation if the VaR can be derived analytically using properties of the distribution(s). It is called parametric since instead of reading the quantile from the empirical distribution, as in historical VaR, the parameters are estimated by e.g. the standard deviation.

Equation 2.1 shows a simple description of how parametric VaR can be calculated. The market price is the value of the portfolio on the market and the volatility can be expressed as the standard deviation multiplied with a confidence level multiplier.

\[
    \text{VaR} = \text{MarketPrice} \times \text{Volatility} \tag{2.1}
\]

As an example say that we have a portfolio with market value $10000, a standard deviation of 1% per day and want a confidence level of 95. If we assume a normal distribution this would mean that the confidence level multiplier is 1.65. This gives us Equation 2.2.

\[
    $10000 \times 0.01 \times 1.65 = $165 \tag{2.2}
\]

This means that with 95% probability the maximum loss will not exceed $165 on one day.

As this example shows parametric VaR can be simple to calculate. The tricky part with it is to make a realistic assumption on the distribution(s), both of the risk factor outcomes and their possible correlation structure. If the assumption is unrealistic the VaR will also be unrealistic.
2.3 Monte Carlo Simulation

Another way to calculate VaR is by using Monte Carlo simulations. This is the most powerful way to calculate VaR [3]. It is very flexible as it can account for a wide range of risks. Monte Carlo simulations can also account for complex pricing patterns and non-linear exposures to the risk factors. These advantages come at a cost though. Both in an intellectual challenge to develop the simulation system and in computing power. The Monte Carlo simulations are more demanding than simpler methods. A simple Monte Carlo simulation is described in Section 2.3.1.

2.3.1 Simulating with one random variable

A simple case of a Monte Carlo simulation is to run the simulations with only one random variable [3]. To get good simulations the choice of stochastic model for the price behavior is important. The geometric brownian motion model is a common model for VaR simulations. The movement steps of the price for the geometric brownian model can be seen in Equation 2.3.

\[
\frac{dS_t}{S_t} = \mu dt + \sigma dz
\]

(2.3)

Here \( S_t \) is the current price and the movement of the price is denoted \( dS_t \). The random variable is \( dz \) which is normally distributed with the variance of \( dt \). Parameters \( \mu \) and \( \sigma \) account for instantaneous drift and volatility at time \( t \). For simplicity they are sometimes assumed to be constant but it is possible to use time varying parameters as well.

We want to get random variables over the VaR time horizon \( \tau = T - t \), where \( t \) is the present time and \( T \) is the target time. In the next step we therefore approximate very tiny increments of \( dt \) with discrete moves of size \( \Delta t \). Now we can integrate over a finite interval with

\[
\Delta S_t = S_{t-1}(\mu \Delta t + \sigma \epsilon \sqrt{\Delta t})
\]

(2.4)

where \( \sigma \) and \( \mu \) are the same as before. \( \epsilon \) is a random variable with mean zero and unit variance.

By generating a sequence of \( \epsilon \)'s, \( \epsilon_1 \ldots \epsilon_n \), we can start to simulate the price path for \( S \). Starting at \( S_t \) we get

\[
S_{t+1} = S_t + S_t(\mu \Delta t + \sigma \epsilon_1 \sqrt{\Delta t})
\]

(2.5)

and \( S_{t+2} \) is computed similarly though by replacing \( t \) with \( t+1 \) and \( \epsilon_1 \) with \( \epsilon_2 \). These indexes increases for each step in the simulation. When all \( n \) steps are simulated we have our price path with the final price of \( S_n \).

With the simulated price path it is possible to simulate the portfolio distribution. This is done by calculating the portfolio value with the final prices of the simulated price path.
This whole process can be repeated as many times as needed, perhaps 10000 times. Finally we can sort the achieved distribution of portfolio values and get the desired quantile based on the wanted confidence level.
Chapter 3

Processing Units

In this chapter the different processing units are described along with their strengths and weaknesses.

3.1 Multicore CPU

The CPU (Central Processing Unit) is the most common type of processing unit for general calculations. Normally this is the unit that is responsible for all, or at least the majority of, the general calculations in a computer system. Server systems for high performance computing can be built to run the majority of calculations on other kinds of processing units depending on their purposes. To achieve high throughput on certain calculations you might want to consider GPUs or coprocessors as your main processing units for these calculations.

3.1.1 Architecture

Multicore CPUs have a few cores that are optimized for high speed. Since the Intel Xeon E5-2697v2 is the CPU in one of the test systems it is used as example. It has 12 cores that can run at up to 3.5 GHz each [4]. Each core can run two hardware threads, meaning the CPU in total can run up to 24 concurrent threads and as the system has 2 CPUs it can run 48 threads concurrently in total. Each core has 32 KB of instruction cache, 32 KB of L1 data cache and 256 KB of L2 cache [5]. The last level cache is 30 MB, giving each core up to 2.5 MB per core of instruction/data last level cache. Compared to GPUs these are large caches. Larger caches means that it is easier to avoid having to access main memory which will improve speed as it is a lot faster to go to the cache than to main memory.

The Xeon E5-2697v2 also supports 256-bit vector instructions [5] (see Section 3.4). Apart from that it also has branch predictors, translation lookaside buffers etc. to further improve speed.
3.1.2 Programming model

OpenCL

Writing OpenCL for a multicore CPU is similar to writing OpenCL for a GPU, see Section 3.2.6. The difference is mainly that you do not need to consider certain optimization such as using local memory as the multicore CPU have caches, which GPUs usually do not.

Java8

With the release of Java8 interesting new features are introduces in Java. Lambdas and parallel streams are two of these that are very interesting in the scope of this thesis. These additions makes it easier to write parallel algorithms in Java.

Lambdas are used to represent a method interface using an expression [6]. With lambdas you can write anonymous inner classes in a more clear and concise way than before. These anonymous classes are often used when you have a class that is only used once in the code, such as an event listener for a specific GUI button. With lambdas the code for doing this gets a lot shorter. In Figure 3.1 you can see short examples of how lambdas can be used.

```java
// Input two integers and return the sum
(int x, int y) -> x + y

// Print a String
(String s) -> {System.out.println(s);}

JButton testButton = new JButton('Test Button');

// Action listener with anonymous class
testButton.addActionListener(new ActionListener(){
    @Override
    public void actionPerformed(ActionEvent ae){
        System.out.println("Click Detected by Anonymous Class");
    }
};

// Action listener using lambda
testButton.addActionListener(e ->
    System.out.println("Click Detected by Lambda Listener");
);
```

Figure 3.1: Lambda examples

Parallel streams is another neat feature introduced in Java8. It simplifies parallelizing your code by allowing the programmers to simply specify that the calculations in the streams shall be performed in parallel [7]. In Figure 3.2 you can see a
3.1. MULTICORE CPU

A short example of vector addition using parallel streams. In the example a lambda expression is also used taking the number of the stream as input and using it as index when adding the two vectors.

```java
int[] vectorA = {1, 2, 3, 4, 5};
int[] vectorB = {11, 12, 13, 14, 15};
int[] result = new int[vectorA.length];

// Make as many streams as there are elements in the vectors
// Run the streams in parallel
// In each stream add the element of the vectors corresponding to the number of the stream
IntStream.range(0, vectorA.length).parallel().forEach(
    index -> {
        result[index] = vectorA[index] + vectorB[index];
    });
```

Figure 3.2: Vector addition with parallel streams

Thus by combining the parallel streams with a lambda expression we have made a parallel algorithm for vector addition with only a few lines on code.

**Fork/Join Framework**

The fork/join framework in Java allows programmers to more easily write parallel programs [8]. Though it demands the programmer to specify how the problem is partitioned. One example on how to use Fork/Join is to implement parallel vector addition. To do this you can write one method that perform the actual addition of elements of two arrays. Though this is only done on small partitions of the arrays. You write another method that checks how large the arrays are and if they are small enough it performs the addition. Otherwise it splits the arrays into smaller partitions and make recursive calls to itself with these new partitions. By calling this recursive function with a ForkJoinPool the recursive calls will be run on different threads thus making the algorithm parallel. Figure 3.3 shows high level pseudo code for how to implement a parallel algorithm with the Fork/Join framework.

**Thread Pool**

A Java Thread Pool is a collection of worker threads that can be used to execute tasks [9]. By giving tasks to the pool it will automatically assign each task to one of the worker threads who will start working on the task. The pool can either be setup to start a new worker for each task or to have a constant number of workers.
calculate (work){
    if the work is small enough
        perform the calculation directly
    else
        split the work into smaller pieces and call calculate
        on every piece and let a ForkJoinPool
        run the calls on separate threads
}

Figure 3.3: Fork/Join pseudo code

By having a constant number of workers you can make sure there will not be more
threads running than the hardware can handle.

Both the parallel features in Java8 and the Fork/Join framework are based on
thread pools. Therefore it is worth noting that it is possible to achieve the same
performance by writing a parallel implementation using a thread pool in Java7 as
by using the Java8 features or the Fork/Join framework. The advantage of Java8
and Fork/Join lies not in performance but in simplicity. As both the Java8 features
and the Fork/Join framework uses thread pools only one of them was chosen for
testing in this thesis, namely Java8.

3.1.3 Strengths and weaknesses

If you have a problem where there is only a little data but a lot of different calcu-
lations on the data the CPU is best as it is fastest at each individual instruction.
Therefore multi-core CPUs are also great at task parallelism, which is when you
can separate different tasks from your problem and run each task in parallel.

3.2 GPU

The GPU (Graphics Processing Unit) has historically been used mainly for graphics,
i.e. to calculate which color each pixel on the screen should have. More recently
the high throughput of GPUs has been used for more general purpose calculations
as well. Realizing this need GPU producers have begun developing GPUs focusing
more on general purpose computing rather than graphics.

3.2.1 NVIDIA Kepler Architecture

The architecture of GPUs differ between vendors and also between their different
models. Since the evaluation will be done on a NVIDIA Tesla K20c GPU the
Kepler architecture will be described in this section as Tesla K20c is based on that
architecture. Kepler is NVIDIA’s latest architecture though it is still quite similar
to the predecessor Fermi used in the other test system.
3.2. GPU

Figure 3.4: Streaming multiprocessor in NVIDIA Kepler [1]
GPUs are optimized for high throughput, meaning the aim is to get as much as possible done at the same time rather than doing every single task as fast as possible. Therefore GPUs have a large amount of cores, though they are generally slower than those on a CPU. On GPUs for general purpose calculations there are two kinds of cores that are used for mathematical calculations, single-precision cores and double-precision cores. As the name suggest the single-precision cores handle single-floating point operations while the double-precision cores handle double-precision floating point operations. These cores are grouped together into something called streaming multiprocessors. In Figure 3.4 you can see a picture of the streaming multiprocessor used in NVIDIA:s Kepler GPUs. In these streaming multiprocessors there are three times as many single-precision cores as double-precision cores, 192 vs 64. Thus it can achieve higher throughput on single-precision calculations. In Figure 3.5 you can see an overview of the Kepler architecture on a GPU with 15 streaming multiprocessors. The Kepler K20c GPU used in the evaluation has 13 streaming multiprocessors, not 15 as in the figure.

Apart from the single- and double-precision cores the streaming multiprocessor can also contain other units with the purpose of speeding up certain calculations. In the Kepler there are 32 load/store units, 32 special function units and 16 texture units [1]. As the naming of the different cores and units would suggest they are all specialized and used for one (or a few) special tasks. Kepler also has four warp
3.2. GPU

Figure 3.6: The Kepler memory hierarchy[1]

Schedulers that schedules threads in groups of 32 threads, called warps.

### 3.2.2 Memory architecture

Not all GPU architectures utilize caches and those that do usually have quite small caches. In Kepler each streaming multiprocessor has an instruction cache, a 48KB read-only cache and 64KB of on board memory. This on board memory is split up into L1 cache and shared memory that is local to this multiprocessor. The shared memory is shared between all cores in the streaming multiprocessor. The split can be configured as 48KB/16KB, 32KB/32KB or 16KB/48KB depending on the need of the application.\(^1\) All streaming multiprocessors also share a bigger L2 cache which is shown in Figure 3.5. The L2 cache is 1536KB. In Figure 3.6 you can see a graphical representation of the memory hierarchy. The figure shows the three layers of memory, with increasing access times the further down you get in the picture. Fastest are the different on-board memories, then the shared L2 cache and lastly the DRAM. The Kepler architecture can have a DRAM of 5-12GB depending on the specific model. Our evaluation model has a DRAM of 5GB.

The GPU is connected with the host system via PCI Express (PCIe) meaning all data going to and from the GPU has to go over the PCIe bus. This adds an extra step in applications using GPU kernels as the applications have to copy data to and from the device. Even though the memory on the GPU is fast, copying data between the host and the device is slow as the PCIe is slow.

### 3.2.3 Programming model

Programs that are run on the GPU are called kernels. When writing a kernel you write the behavior of one single thread. Then all threads in the kernel execute these

\(^1\)At the moment this is only configurable in CUDA and not in OpenCL.
same instructions. So the base case is that you divide the data to process based on the thread id, e.g. use the id as index into a vector. This way all threads can execute the same instructions but on different data.

Each thread in a kernel is run on a separate core. The threads are grouped into work groups (OpenCL) or blocks (CUDA). Each streaming multiprocessor can run one or more of these work groups at once. Each work group can only perform a single instruction at the same time even if it is running on multiple cores. The data on which the operations are done on can be different, this is called the SIMD (Single Instruction Multiple Data) model. For data-parallel problems this is a good model since you have a lot of data on which you want to perform the same instruction. If you instead have problems which are not very data-parallel this is not a good model. As an example, consider a program that has an if-statement run in a work group with two threads and one of the threads enters the if-statement while the other enters the else-statement. This would mean that while the first thread does the work inside the if-statement the second thread is idle. When the first thread is finished it would then be idle while the second thread does its work in the else-statement. This could lead to a lot of cores being idle instead of doing valuable work.

The flow of an application utilizing the GPU by OpenCL could be as follows:

1. Initialize everything needed for running on the device.
2. Copy data to device.
3. Set arguments for the kernel.
4. Execute the kernel.
5. Copy result back to host.
6. Clean up the memory.

All steps between 2 and 5 have to be performed every time you want to run the kernel within the application. Depending on the problem and the data size the transfer to and from the device can actually take more time than the execution of the kernel.

3.2.4 Strengths and weaknesses

Since GPUs are focused on high throughput they excel at computationally intensive programs with high data parallelism. The less computations per data the worse the performance of the GPU gets as the overhead of the data transfer will be bigger. Branches are not good for performance on the GPU as it will make parts of the hardware idle while there is still work left to do.

3.2.5 Programming Languages

There are two main ways to utilize GPUs for calculations in programs. The first is to write your own GPU kernels in one of the available programming languages OpenCL or CUDA. Option number two is to make use of some higher level framework like
Aparapi (OpenCL) or Rootbeer (CUDA). These frameworks let the programmer run programs on the GPU with less knowledge and effort than to program your own kernels. Aparapi might not able to parallelize as advanced scenarios as writing CUDA or OpenCL directly, nor is it possible for the programmer to optimize the GPU code.

If you compare CUDA with OpenCL the biggest difference is that CUDA only supports GPUs from NVIDIA while OpenCL supports multiple brands of GPUs. Worth noting is that while OpenCL supports different hardware the code might still have to be re-optimized with each change of hardware. OpenCL also supports coprocessors, like the Intel Xeon Phi, and multicore CPUs apart from GPUs. The advantage CUDA has is that it has a more centralized development tool while in OpenCL you have to rely on different tools for different hardware vendors. Earlier research appears to find them to perform equally after optimizing the code [10]. The approach to optimizing the code appears to be different in the two languages which can give differences in performance when comparing the two languages if you do not optimize both codes according to these differences. In this thesis only OpenCL is examined as it is of interest to be as platform independent as possible.

### 3.2.6 OpenCL

The basic OpenCL kernel uses each threads id to partition the calculation. Using vector addition as example you write a kernel like the one in Figure 3.7. First it reads the id of the thread and then uses this id to determine which elements this thread shall perform the calculation on. For this to work the kernel must be called with as many threads as there are elements in the two arrays. This approach work good when the hardware can run a large number of threads at once as there are many threads that only perform a small calculation each.

```java
kernel void vectorAddition (int [] arrayA, int [] arrayB, int [] result)
{
    // Get the id
    int id = get_global_id(0);
    // Multiply one elements based on the id
    result[id] = arrayA[id] + arrayB[id];
}
```

Figure 3.7: Kernel for vector addition in OpenCL

### Language bindings for Java

There are a couple of different language bindings that can be used to call OpenCL kernels from within Java. Three examples are JOCL [11], JavaCL [12] and JOCL from JogAmp [13]. The main difference appears to be that both JOCL and JogAmps
JOCL uses JNI to call the kernels that are written outside of Java while JavaCL uses BridJ. JogAmps JOCL and JavaCL both provide a object oriented abstraction of OpenCL while JOCL is very close to the original OpenCL API [11, 12, 13]. An example of how to use JogAmps JOCL can be seen in Figure 3.8.

```java
public class JogampsJOCLVectorAddition {
    // Member variables
    int[] mArrayA, mArrayB, mResult;
    CLBuffer<IntBuffer> mBufferA, mBufferB, mBufferC;
    int mLocalworksize;
    // Get the required information about the system
    CLPlatforms platforms = CLPlatform.listCLPlatforms();
    CLContext context = CLContext.create(
            platforms[0].listCLDevices()[0]);
    // Get the device with highest FLOPS
    CLDevice device = context.getMaxFlopsDevice();
    CLCommandQueue queue = device.createCommandQueue();
    CLProgram program = context.createProgram("filename.cl");
    CLKernel kernel = program.createCLKernel("kernelname");
    
    public void vectorAddition(){
        // Init the buffers
        mBufferA.getBuffer().put(mArrayA);
        mBufferB.getBuffer().put(mArrayB);
        mBufferC.getBuffer().put(mResult);
        // Set the arguments for the kernel
        kernel.setArg(0, mBufferA);
        kernel.setArg(1, mBufferB);
        kernel.setArg(2, mBufferC);
        // Copy the arrays to the device, wait for last copy
        // to finish before continuing
        queue.putWriteBuffer(mBufferA, false)
            .putWriteBuffer(mBufferB, true);
        // Start the calculation and read the result
        queue.put1DRangeKernel(kernel, 0, mArrayA.length, mLocalworksize)
            .putReadBuffer(mBufferC, true);
        // Put the result in the member result
        mBufferC.getBuffer().get(mResult);
    }
}
```

Figure 3.8: Starting an OpenCl kernel in Jogamp's JOCL
3.2. GPUs

3.2.7 Aparapi

Aparapi is an experimental high level parallel API for Java [14]. It contains methods to offload work to be run on a GPU or multicore CPU. It allows programmers to write their kernels in Java code and then generates the OpenCL code at runtime and executes it on the specified device (GPU or multicore CPU). This makes it easier for programmers not familiar with OpenCL or GPU programming to run programs on the GPU. It also comes with some limitations though. Since Aparapi generates the OpenCL code you cannot make your own optimizations to that code. If you are unfamiliar with OpenCL but still want to try to write your own optimized OpenCL code you can start by generating the code with Aparapi and then modify it and run it with one of the language bindings instead. Aparapi provides a way to retrieve the generated code so that it is possible to look at what is actually run on the GPU.

When writing an Aparapi kernel you write a class that extends the Kernel class from the Aparapi package. This class must implement the method run, which is the actual kernel that will be generated into OpenCL code. Instead of adding parameters to this method all variables that need to be accessed in the kernel must be declared as member variables in the class. Worth noting is that Aparapi only provides support for primitive data types at the time of writing. Thus you cannot take advantage of objects as you are used to in Java.

```java
public class AparapiVectorAddition extends Kernel{
    // member variables
    int [] mVectorA, mVectorB, mResult;

    // This is the kernel that gets generated to OpenCL
    @Override
    public void run(){
        int id = getGlobalId();
        mResult[id] = mVectorA[id] + mVectorB[id];
    }

    // This method runs the vector addition
    public void vectorAddition(){
        // Set execution mode (GPU, CPU or JavaThreadPool)
        setExecutionMode(EXECUTION_MODE.GPU);
        // Execute with as many threads as there are elements
        // in the arrays
        execute(mVectorA.length);
    }
}
```

Figure 3.9: Vector addition in Aparapi
To run Aparapi kernel you write a method that specifies which execution mode should be used, i.e. which device to run on. You can specify to run either on the GPU or on the CPU. A third option exists, to run using a Java thread pool. This can be used as a fallback if the program is run on a machine that does not have OpenCL support. Lastly when executing the kernel you must specify the size of the problem, just like in OpenCL and just like in OpenCL the range can be expressed in one or two dimensions depending on how you want to partition your problem.

Figure 3.9 shows an example of how you can write vectors addition using Aparapi. The code inside the run method is very similar to the OpenCL kernel in Figure 3.7. Though with Aparapi the code for copying the arrays to the device is generated automatically.

There is a project called Sumatra that is working on enabling the Java Hotspot JVM to run certain parts of the JVM on GPUs and later also support application programmers to run their code on GPUs [15]. This is an ongoing project that has not yet been released and thus not tested in this thesis.

### 3.3 Coprocessors

Another way to speed up parallel programs is to use a coprocessor. These are processing units that are made to be good at handling computationally intensive programs. They can be used as a supplement to the CPU, just like a GPU. The difference from a GPU is that a coprocessor can consist of multiple CPU cores with large caches and branch predictors etc. Since it is similar to the multicore CPU you can run the same kind of programs as you can run on a CPU. This means it is a lot easier for most programmers to program for a coprocessor compared to a GPU which has a totally different architecture and needs very specialized code.

#### 3.3.1 Intel Xeon Phi Architecture

The Intel Xeon Phi has 57 cores and each core runs at 1.1 gHz and can run 4 hardware threads each for a total of 228 threads. Each core has a L1 cache of 32KB and a L2 cache of 512KB. You can see an overview of a core in Figure 3.11. L2 caches are kept fully coherent with a global-distributed tag directory [2]. This means that even though different cores change the same value from memory the caches will be updated to not show any old values. The size of the DRAM is 5.8 GB and thus slightly larger than on the Kepler GPU. Like GPU the coprocessor is connected via PCIe.

Each Xeon Phi core also supports vector instructions of 512-bits. Thus it can handle 16 single-precision or 8 double-precision floating point operations at once in each thread.
3.3. COPROCESSORS

Figure 3.10: Architecture of the Xeon Phi [2]

Figure 3.11: Architecture of a core in the Xeon Phi [2]
3.3.2 Strengths and weaknesses

The coprocessor is better at computationally intensive programs than the CPU while still having familiar hardware that makes it easier to program than the GPU. It might not offer the same throughput as a GPU at truly data parallel problems and since the clock rate is slower than that on a CPU it is slower than the CPU on serial programs. As the coprocessor is connected via PCIe it also suffers from the same overhead of data transfer as the GPU.

3.4 Vector instructions

A third way to utilize the data-parallelism of the VaR algorithm is to use vector instructions. With these instructions you can do multiple calculations for each instruction. This way the total number of instructions needed is reduced, which in turn means that the execution time of the algorithm is reduced. In the evaluation system both the CPU and coprocessor can utilize vector instructions.

To use vector instructions the hardware must have a register that is larger than the normal registers and can contain multiple values of the data type used. The hardware can then perform the same operation on all the values in that register.

3.4.1 Strengths and weaknesses

Vector instructions can only give a maximum speedup linear to the number of values it can fit into the register and perform operations on. Another limitation is that it only works when you have data parallel problems where you execute the same instruction on different data.

A strength is that it is easy to apply on all places in the code where you run the same instruction on multiple data. This way it is easy to get a speedup on these parts without any change at all in the algorithm.
Part II

Results
Chapter 4

Implementation

In this chapter some general details around the problem and the different implementations in the thesis will be described.

4.1 VaR Algorithm

The algorithm implemented is a scenario based algorithm using an assumption of linear risk factor exposures. In the algorithm there is a matrix where each column is a risk factor and each row is a scenario. The value in each cell represents the shift in value of the risk factor in that scenario e.g. in one scenario the value of an equity is -$10. This matrix is then multiplied with a portfolio represented as a vector where each cell is the quantity owned of a specific asset. The result is a Profit & Loss (P&L) vector containing the P&L of the portfolio in each scenario. From this P&L vector you can then calculate the VaR value, e.g. by taking the P&L for the second worst scenario.

This is a data parallel problem as each element in the P&L vector can be calculated independent of each other. Thus this is a calculation that theoretically could perform well on a GPU or coprocessor as their strength is data parallel problems.

You can see an example calculation of a P&L vector in Table 4.1-4.4. The scenario matrix in Table 4.1 is multiplied with the portfolio vector (matrix multiplication) in Table 4.2. The multiplication can be seen in Table 4.3 while the resulting P&L vector is shown in Table 4.4.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>RF 1</th>
<th>RF 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>−$1</td>
<td>$2</td>
</tr>
<tr>
<td>2</td>
<td>$2</td>
<td>−$1</td>
</tr>
<tr>
<td>3</td>
<td>$2</td>
<td>$2</td>
</tr>
</tbody>
</table>

Table 4.1: Scenario matrix with 3 scenarios and 2 risk factors
CHAPTER 4. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Asset</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.2: Portfolio with 2 exposures

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((-1 \times 1) + (2 \times 2))</td>
</tr>
<tr>
<td>2</td>
<td>((2 \times 1) + (-1 \times 2))</td>
</tr>
<tr>
<td>3</td>
<td>((2 \times 1) + (2 \times 2))</td>
</tr>
</tbody>
</table>

Table 4.3: Matrix-vector multiplication

<table>
<thead>
<tr>
<th>Scenario</th>
<th>P&amp;L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$3</td>
</tr>
<tr>
<td>2</td>
<td>$0</td>
</tr>
<tr>
<td>3</td>
<td>$6</td>
</tr>
</tbody>
</table>

Table 4.4: Resulting Profit & Loss vector

4.2 Properties of the problem

The VaR-algorithm used in the thesis is essentially a matrix-vector multiplication. Input to the problem is therefore a matrix and a vector. For the matrix vector multiplication to work the number of columns in the matrix must be equal to the length of the vector. Apart from that there are no theoretical restrictions on how large or small the matrix and vector can be for the multiplication to work. Since the thesis is testing VaR and not general matrix vector multiplication the matrix will most likely have a larger number of columns (risk factors) than rows (scenarios). Thus we have some knowledge about the problem beforehand and the implementations and tests will be done taking this knowledge into consideration.

4.2.1 Parallelizing the calculation

There are basically two ways to make the matrix vector calculation parallel. The first and most straightforward way is to calculate each row in parallel.

For the second way we want to find a way to increase the parallelism even further. Since there will not be a huge number of rows in the matrix it is interesting to split up each row into multiple calculations that can be done in parallel. This way the parallelism of the algorithm can be increased which theoretically gives us the possibility of an even greater speedup. As all elements in each row will be multiplied with a separate value in the vector each multiplication can be done in parallel without interfering with each other. Once the multiplications are finished it is time to sum all the resulting products. If the multiplications are done in parallel
4.3. SERIAL REFERENCE VERSION

synchronization is needed to make sure all calculations are finished before starting to sum the products. The sum cannot be calculated completely in parallel as each value has to be added together, thus it can at most be calculated parts at a time and then the final sum has to be calculated serially.

Instead of having threads performing only one calculation it is possible to let each thread multiply a “chunk” of elements on each row instead of just one element. As an example we have a matrix where every row contains four elements. We can let one thread multiply the first and second element with the first and second value in the vector respectively and have another thread take care of the remaining two elements. An illustration of this example can be seen in Figure 4.1. It is also possible to let the same thread calculate the sum of all its calculated products and finally have one thread calculate the sum of all these partial sums. By choosing how many pieces to split each row into it is possible to manage the total number of threads and to be able to utilize the hardware in the best way.

Figure 4.1: One way to split the multiplication of a row. Each cell represents a single element.

4.3 Serial reference version

As the speedup should be calculated compared to the best serial version, two different serial algorithms were tested. The first is a naive version which simply does matrix vector multiplication by multiplying each row with the vector, one after the other.

The second version is a tiled (or blocked) matrix multiplication. In this version the matrix and vector is split up into tiles (or blocks). Now the matrix can be viewed as a matrix of blocks that can be multiplied with an algorithm that is similar to the naive version. For each row of blocks all blocks are multiplied with the corresponding block in the vector by using the naive version. The resulting products for each row are summarized to get the final result. Example code for this version can be seen in Figure 4.2.

By tiling the matrix it is possible to take maximum advantage of the caches. The goal is to make the tiles large (or small) enough so that one tile from the matrix, the corresponding tile from the vector and their result tile fits in the cache at the same time. This way the number of cache misses is minimized. This version should perform better than the naive version when the problem size gets larger as the naive version will get reduced performance due to cache misses.
public void tiledMult(double[] pMatrix, double[] pVector,
  double[] pResult, int pRiskFactors){
  // For each row of tiles multiply each tile separately
  for (int y = 0; y < BLOCKS_Y; y++) {
    double tBlockRes = new double[SIZE_Y];
    for (int x = 0; x < BLOCKS_X; x++) {
      normalMultiply(pMatrix, pVector, pRiskFactors, y, x,
                     tBlockRes);
    }
    // Store the result for each row
    int row = y * SIZE_Y;
    for (int i = 0; i < SIZE_Y; i++) {
      pResult[row + i] = tBlockRes[i];
    }
  }
}

public void normalMult(double[] pMatrix, double[] pVector,
  int pRiskFactors, int pY, int pX, double[] pRes){
  // For each row in the tile multiply it with the
  // corresponding part of pVector
  int tRow = pY * SIZE_Y;
  for (int i = 0; i < SIZE_Y; i++) {
    double tSum = 0;
    for (int k = 0; k < SIZE_X; k++) {
      int tCol = pX * SIZE_X + k;
      tSum = pMatrix[(tRow + i) * pRiskFactors + tCol] * pVector[tCol];
    }
    pRes[i] += tSum;
  }
}

Figure 4.2: Tiled serial implementation

4.4 GPU

In this section the different versions implemented on the GPU are described.

4.4.1 Different versions

All implementations for the GPU are either written directly in OpenCL or in Java
and then the OpenCL code is automatically generated by Aparapi. Section 4 shows
which implementations were done in OpenCL and Aparapi.
4.4. GPU

Full rows

The most intuitive way to implement matrix vector multiplication on a GPU is the first one described in Section 4.2.1. In this version the work is split up so that each thread calculates one row multiplied with the vector and then sum all the products. An OpenCL kernel example for this algorithm can be seen in Figure 4.3. When calling the OpenCL kernel you specify a total number of work items equal to the number of rows. This way each thread can use its unique thread id to decide which row to calculate as there are the same number of rows and threads.

```
kernel void mult( global const double* pMatrix,
                 global const double* pVector,
                 global double* pResult,
                 int pScenarios,
                 int pRiskFactors){
    // get index of thread
    int row = get_global_id(0);
    // multiply row with vector
    double tValue = 0;
    for (int k = 0; k < pRiskFactors; k++) {
        tValue += pMatrix[k + row * pRiskFactors] * pVector[k];
    }
    pResult[row] = tValue;
}
```

Figure 4.3: OpenCL kernel where each thread calculates a whole row

Partitioned rows

If there are more cores on the GPU than there are rows in the matrix there will be some cores being idle. To better utilize the hardware and achieve better performance other versions of the VaR algorithm were also implemented where the rows of the matrix are split into multiple parts that are calculated in parallel. When all partitions are calculated the result for each partition on a row is summed and becomes the final result for that row, an example kernel for doing this final sum can be seen in Figure 4.5.

This should perform better than using full rows when the problem size gets large enough. With a small problem there will likely be too much overhead from the extra work compared to using full rows.

The partitioning of the row can be done in multiple ways. The first way is to let each thread calculate a part of a single row and have a work group containing all threads needed to calculate one single row. This is illustrated in Figure 4.4b. Example code for this version can be seen in Figure 4.6.
CHAPTER 4. IMPLEMENTATION

A second way is to have each thread calculate part of a row but having the work group contain multiple threads calculating the same columns but on different rows. Figure 4.4c illustrates this way of partitioning the matrix. Partitioning like this should allow for maximum utilization of each streaming multiprocessors local memory as you can split the row into as large parts as fits into the local memory and then have multiple threads utilizing the values from the local memory. Example code of how this partitioning is done can be seen in Figure 4.8.

![Diagram showing three different ways to partition calculations](image)

Figure 4.4: Three different ways to partition the calculation.

```c
kernel void sum_rows(global double* pResult,
                     int pNumRows,
                     int pThreadsPerRow)
{
    // Get the row to sum
    int row = get_global_id(0);
    double sum = 0;
    // Sum all partial sums
    for (int col = 0; col < pThreadsPerRow; col++) {
        sum += pResult[row + pNumRows * col];
    }
    // Store the final result in the first element of the row
    result[row] = sum;
}
```

Figure 4.5: OpenCL kernel for summarizing partial results from partitioned rows

A third possible way is to have each thread calculate one partition but on multiple rows. Using Figure 4.4c as illustration this would mean that each group is calculated by a single thread. As the matrix has few rows this would not increase parallelism and therefore in theory not give better performance. Also this has already been tested by [16] and therefore this version was not tested in this thesis.
4.4. GPU

kernel void mult(global const double* pMatrix, 
    global const double* pVector, 
    global double* pResult, 
    int pScenarios, 
    int pRiskFactors){

    // get number of risk factors to compute
    int tActualRFs = pRiskFactors / 
        get_global_size(RISKFACTORS_DIM); 
    // Get the first risk factor
    int tStart = tActualRFs * get_global_id(SERIES_DIM); 
    // Get the final risk factor
    int tEnd = tActualRFs + tStart; 

    // Calculate the partial dot product
    for (int tRF = tStart; tRF < tEnd; tRF++){
        sum += pMatrix[tRF + pRiskFactors * 
            get_global_id(SCENARIOS_DIM)] * pVector[tRF]; 
    }
    // Store in pResult
    pResult[get_global_id(SCENARIOS_DIM) + pScenarios * 
        get_global_id(SERIES_DIM)] = sum;
}

Figure 4.6: OpenCL kernel where each thread calculates a partitioned row as in Figure 4.4b

4.4.2 Local memory

As written in Section 3.2 the Kepler GPU has local memory that is shared within a streaming multiprocessor. Access to this local memory is faster than access to the larger globally accessible memory. Therefore versions that take advantage of the local memory were implemented. Since this local memory cannot be accessed by the host the data must first be transferred to the global memory on the device and then copied by the kernel to the local memory. Since the copy will require extra instructions compared to just using global memory both versions using only the global memory and versions using local memory were implemented for comparison.

When using local memory the matrix has to be partitioned in such a way that all elements that are to be calculated from the vector fit into the local memory of the streaming multiprocessor. This puts a restriction on how many columns each work group can calculate. Since all rows are multiplied with the same vector it is a better choice to store the vector in local memory as there will be more accesses to those values. In Figure 4.7 is an example of a kernel that utilizes local memory.
CHAPTER 4. IMPLEMENTATION

```c
kernel void mult( global const double* pMatrix,
                 global const double* pVector,
                 global double* pResult,
                 int pScenarios,
                 int pRiskFactors
                 local double* pLocal){
    // Copy risk factors to local memory
    // If fewer risk factors than threads copy one each
    if(pRiskFactors <= get_work_dim()){
        if(get_local_id(0) < pRiskFactors){
            pLocal[get_local_id(0)] = pVector[get_local_id(0)];
        }
    }
    // Else every thread has to copy multiple values
    else{
        int tNum = 1 + pSeries / get_work_dim();
        for(int i=0; i < tNum; ++i) {
            int tRF = get_local_id(0) * tNum + i;
            if(tRF < pSeries){pLocal[tRF] = pVector[tRF]; }
        }
    }
    // Make sure all threads in work group are finished
    barrier(CLK_LOCAL_MEM_FENCE);
    // Do the calculation
    int row = get_global_id(0);
    double sum = 0;
    for(int i = 0; i < pSeries; i++){
        sum = pMatrix[i + row * pRiskFactors] * pLocal[i];
    }
    pResult[row] = sum;
}
```

Figure 4.7: OpenCL kernel where each thread calculates a full row and utilizes local memory

4.4.3 Vectors

The GPU can use two special data types. These are called double# and float#, where # is a number from 2 up to 8 for double i.e. double8 and up to 16 for float i.e. float16. These are vectors with up to 8 doubles or up to 16 floats. To utilize these vectors the same algorithm as with single values can be used. With single values the algorithm is to multiply each value in every row with the corresponding element in the vector and the summarize all of the received products for each row.
Instead of doing this for each value you can do it with 8 values at the time if you use double8. Each multiplication will then give a product of the data type double8. So when each row is calculated there is one additional step to the normal algorithm, you have to summarize each value in the double8 result for that row. This final sum is then the same as the sum in the normal algorithm.

Multiple versions were implemented with the possibility to choose which of these vector data types to use in the algorithm, or to use normal floating points. One example is shown in Figure 4.8 which uses vector data types.

```c
kernel void mult( global const double8* pMatrix,
                 global const double8* pVector,
                 global double* pResult,
                 int pScenarios,
                 int pRiskFactors)
{
    // Size of the vector datatype
    int tSize = 8;
    // Number of vectors with risk factors
    int tRFs = pRiskFactors / tSize;
    int tRFPerThread = pSeries / get_global_size(RF_DIM);
    // Actual number of risk factors to calculate
    int tActualRFs = tRFPerThread / tSize;
    int tStartRF = tActualRFs * get_global_id(RF_DIM);
    int tEndRF = tActualRFs + tStartRF;
    int tScenario = get_global_id(SCENARIO_DIM);
    int tScenBase = tScenario * tRFs;
    double8 tSumVec = (double8) 0;
    // Do the multiplication with vectors
    for (int i = tStartRF; i < tEndRF; i++){
        tSumVec += pMatrix[i+tScenBase] * pVector[i-tStartRF];
    }
    // Sum all values in the vector into one final sum
    double* tSumPtr = &tSumVec;
    double tSum = 0;
    for (int i=0; i < tSize; ++i){
        tSum += tSumPtr[i];
    }
    // Store in pResult
    pResult[tScenario+get_global_id(RF_DIM)*pScenarios] = tSum;
}
```

Figure 4.8: OpenCL kernel where each thread calculates a partitioned row as in Figure 4.4c and utilizing the vector datatype
4.4.4 Language bindings

As stated in Section 3.2.6 the difference between Jogamps Jocl, JavaCL and JOCL lies in the use of either JNI or BridJ. As JNI is supposed to be slightly faster than BridJ to call native kernels [17] the choice stood between Jogamps Jocl and JOCL. Both these versions use JNI but Jogamps Jocl provide an object oriented abstraction of OpenCL while JOCL is made to be very close to the OpenCL specification [11]. Thus Jogamps Jocl was chosen for this thesis as the abstraction simplifies usage compared to JOCL [11].

4.4.5 Aparapi

Multiple implementations were done in Aparapi as can be seen in Section 4.8. One implementation for each of the execution modes: GPU, CPU and Java thread pool and for each execution mode one implementation calculating full rows and one calculating partitioned rows. Example code for how to implement on full rows can be seen in Figure 4.9. To change the execution mode all that has to be done is set the execution mode parameter.

```java
public class AparapiMult extends Kernel{
    double[] mMatrix, mVector, mResult;
    int mScenarios;

    // This is the kernel that is generated to OpenCL
    @Override
    public void run(){
        int tRow = getGlobalId ();
        double tSum = 0;
        for(int tCol = 0; tCol < mRiskFactors; tCol++){
            tSum += mMatrix[tCol+tRow*mRiskFactors]*Vector[tCol];
        }
        mResult[tRow] = tSum;
    }

    // This method runs the multiplication
    public void multiply(EXECUTION_MODE pMode){
        // Set execution mode (GPU, CPU or JavaThreadPool)
        setExecutionMode(pMode);
        // Execute with as many threads as there are scenarios
        execute(mScenarios);
    }
}
```

Figure 4.9: Aparapi kernel where each thread calculates a full row
4.5 Coprocessor

In this section the implementations on the Intel Xeon Phi will be described.

4.5.1 OpenCL

Intel provides an SDK for OpenCL Applications [18] that can be used to write and run OpenCL kernels on their CPUs and coprocessors. The kernels are similar to those for a GPU. Since the underlying hardware is different it might not be the same version that is fastest on the GPU that is also the fastest on the CPU/coprocessor. The Intel SDK gives you vectorization for free [19] and the programmer does not have to care about local memory as the CPUs and coprocessors have caches that take care of minimizing the I/O times automatically. Caches still require the programmer to take data locality into account to avoid cache misses as much as possible though. The code for transferring data to and from the device is the same as with the GPU.

4.6 Vector instructions

To evaluate the performance gain of using vector instructions on the CPU two different versions were implemented. One serial version written in Java and one parallel version written in OpenCL using Intel SDK for OpenCL Applications [18].

In Java there is no way to explicitly tell the processor to use vector instructions. If the JIT compiler understands that it can utilize vector instructions it will do that automatically [20]. Therefore the way to go is to implement the calculation in such a way that the JIT compiler understands that it would benefit from vector instructions.

When inspecting the generated assembly code from the Java JIT compiler it is possible to see that the reference algorithm with a normal double for-loop actually uses a few vector instructions. It only makes use of four of the available wide registers though. When unrolling the for-loop to do 8 or 16 calculations in each round the assembly code shows that the compiler actually makes use of 8 and 16 registers respectively.

The instructions used appear to be SSE instructions, which is no longer the latest set of vector instructions. There is a flag called UseAVX=# which can be used to either tell the compiler not to use AVX or to use AVX1/AVX2. In the tests no difference in performance was found when setting this flag nor any differences in the generated assembly code.

When writing OpenCL with Intel SDK for OpenCL Applications the SDK will automatically identify where it is possible to use vector instructions with less help needed from the programmer than in Java. The compiler will make the program use vector instructions on the calculations based on dimension zero of the work group [19].
CHAPTER 4. IMPLEMENTATION

```java
for (int row = 0; row < NUM_ROWS; row++) {
    int sum = 0;
    // Unrolled loop
    for (int col = 0; col < NUM_COLS; col += 4) {
        int m_index = row * NUM_COLS + col;
        result[row] = matrix[m_index] * vector[col] +
                      matrix[m_index + 1] * vector[col + 1] +
                      matrix[m_index + 2] * vector[col + 2] +
                      matrix[m_index + 3] * vector[col + 3];
    }
}
```

Figure 4.10: How to hint the JIT compiler about vectorization

4.7 Java8

Two versions were implemented with Java8 features. One parallel version of each of the serial reference versions. Both were implemented with parallel streams. In the naive implementation there are two for-each streams, the outer going through all rows and the inner going through all columns of that row in the matrix. Here you have two design choices, either you make the outer loop parallel or the inner one. If you make the inner loop parallel this means the work has to be divided for each round in the outer loop. By making the outer loop in parallel the work only has to be divided once, resulting in a smaller overhead.

In Figure 4.11 you can see a Java8 implementation of the naive algorithm with the outer streams being parallel.

```java
// Run parallel streams for all the rows
IntStream.range(0, NUM_ROWS).parallel().forEach(row -> {
    // get start and end element of the row in matrix
    int start = row * NUM_COLS;
    // get end element of the row in matrix
    int end = start + NUM_COLS;
    double sum = 0;
    // Run serial streams over all columns in the row
    IntStream.range(start, end)
        .forEach(i -> sum += matrix[i] * vector[i - start]);
    // Store result in result vector
    result[row] = sum;
});
```

Figure 4.11: Matrix vector multiplication in Java8
4.8 Implementations

In this section all the different implementations will be presented with a short description. For more details on the different optimizations see the corresponding section in Chapter 4.

**AparapiCPU** Implementation written in Java with Aparapi, running on the GPU. The calculation is performed on full rows.

**AparapiCPUSplit** Implementation written in Java with Aparapi, running on the CPU. The calculation is performed on partitioned rows.

**AparapiGPU** Implementation written in Java with Aparapi, running on the GPU. The calculation is performed on full rows.

**AparapiGPUSplit** Implementation written in Java with Aparapi, running on the GPU. The calculation is performed on partitioned rows.

**AparapiJTP** Implementation written in Java with Aparapi, running on the CPU using a Java Thread Pool. The calculation is performed on full rows.

**AparapiJTPSplit** Implementation written in Java with Aparapi, running on the CPU using a Java Thread Pool. The calculation is performed on partitioned rows.

**CPUOpenCL** Implementation written in OpenCL, running on the CPU. The calculation is performed on full rows and optimized with automatic vectorization performed by the Intel SDK for OpenCL.

**Java8** Implementation in Java that is parallelized using lambdas and parallel streams. In the results only the fastest Java8 version is presented, i.e. only the one based on the naive version or the one based on the tiled version. Runs on the CPU.

**JavaVectorInstructions** Implementation in Java that is written so that the Java JIT-compiler understands that it can benefit from vector instructions and thus compile the code to use vector instructions. Runs on the CPU.

**GPULocal** Implementation written in OpenCL and running on the GPU. The calculation is performed on full rows and utilizing the GPUs local memory for the portfolio vector.

**GPULocalVectors** Implementation written in OpenCL running on the GPU. The calculation is performed on full rows utilizing the vector data type and the GPUs local memory for the portfolio vector.
CHAPTER 4. IMPLEMENTATION

**GPU Local Vectors Split** Implementation written in OpenCL running on the GPU. The calculation is performed on partitioned rows utilizing the vector data type and the GPUs local memory for the portfolio vector. The partitioning of the rows is the first one described in Section 4.4.1 and therefore only works when the whole portfolio vector fits into the local memory.

**GPU Local Vectors Split Better** Implementation written in OpenCL running on the GPU. The calculation is performed on partitioned rows utilizing the vector data type and the GPUs local memory for the portfolio vector. The partitioning of the rows is the second one described in Section 4.4.1.

**GPU Global** Implementation written in OpenCL running on the GPU. The calculation is performed on full rows and utilizing only the GPUs global memory.

**GPU Global Split** Implementation written in OpenCL running on the GPU. The calculation is performed on partitioned rows and utilizing only the GPUs global memory.

**GPU Global Vectors Split** Implementation written in OpenCL running on the GPU. The calculation is performed on partitioned rows, utilizing only the GPUs global memory and utilizing the vector data type.

**Serial** Serial implementation in Java. In the result only the fastest of the two serial versions is presented, i.e. only the naive version or the tiled version. Runs on the CPU.

**Xeon Phi** Implementation written in OpenCL running on the coprocessor Xeon Phi. The calculation is performed on full rows and optimized with automatic vectorization performed by the Intel SDK for OpenCL.

### 4.9 Expectations

In this section the general expectations of the different implementations will be described.

**GPU** Looking at the different GPU implementations in Section 4.8 the one with the most optimizations, GPU Local Vectors Split Better, should perform best given a large enough problem. The optimizations comes with an extra overhead meaning they might decrease performance on smaller problem sizes as the overhead might be bigger than the improvement.

**Aparapi** With Aparapi there is no way to optimize the kernels that are generated by Aparapi. Therefore Aparapi will likely not perform as well as the optimized OpenCL kernels that use local memory and vector data types.
4.9. EXPECTATIONS

**Vector Instructions** Using vector instructions in Java should improve performance compared to running a serial version though it will likely not give as large speedup as the GPU, coprocessor or Java8 versions. Vector instructions can only improve performance with at most a factor as large as the number of values that fit in the vectors.

**Java8** Java8 will likely give a good speedup as the problem is inherently parallel and all cores on the multicore CPUs can be utilized.

4.9.1 Expected winner

The problem is inherently data parallel and should therefore perform well on a GPU and coprocessor that have many more cores than a multicore CPU. This leads to the belief that they should have an advantage compared to the multicore CPU. Though the Java8 implementation also has its advantages. It does not suffer from any data copy to and from the device nor from making a call to a function outside of Java which the GPU and coprocessor do. Therefore it is difficult to know beforehand which implementation will win the performance test.
Chapter 5

Performance tests method

This chapter describes how the tests were performed and on what hardware they were run.

5.1 Hardware

The performance tests will be performed on three different systems. One supplied by SICS and two by Cinnober.

The following systems will be used for performance testing:

- Valarauko: A system with two Intel Xeon E5-2697v2 CPUs [4] that can run 48 hardware threads at 3.5 GHz. One NVIDIA Tesla K20c GPU [1] and one Intel Xeon Phi coprocessor [21] that can run 228 hardware threads at 1.1 GHz.
- Sammy: A system with two Intel Xeon X5690 CPUs [22] that can run 24 hardware threads at 3.73 GHz and one NVIDIA Tesla M2050 GPU [23].
- Bacall: A system with two Intel Xeon E5-2690v2 CPUs [24] that can run 40 hardware threads at 3.6 GHz.

5.2 What to measure

Getting good and trustworthy results from performance tests requires careful preparations. When comparing performance of different applications it is important to make sure the tests are measuring the same thing in both applications. This is not always as straightforward as it sounds. As all applications tested in this thesis were custom made for this purpose it was a bit easier. They could all be designed to have the same structure. When you have applications with different structure it is not always obvious where to start and stop measuring the time.

As this thesis aim to compare performance on GPUs and coprocessors with performance on CPUs it is not only the calculation time that has to be measured. As the GPU/coprocessor cannot access the main memory like the CPU the data
CHAPTER 5. PERFORMANCE TESTS METHOD

<table>
<thead>
<tr>
<th>Processing Unit</th>
<th>What to measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Calculation</td>
</tr>
<tr>
<td>GPU</td>
<td>Transfer of input to device</td>
</tr>
<tr>
<td></td>
<td>Calculation</td>
</tr>
<tr>
<td></td>
<td>Transfer of output back to host</td>
</tr>
<tr>
<td>Coprocessor</td>
<td>Transfer of input to device</td>
</tr>
<tr>
<td></td>
<td>Calculation</td>
</tr>
<tr>
<td></td>
<td>Transfer of output back to host</td>
</tr>
</tbody>
</table>

Table 5.1: What to measure for a fair performance comparison

transfer also has to be measured for the results to be comparable. In Table 5.1 you can see what is measured depending on which hardware is used.

5.3 Performance tests in Java

When doing performance tests in Java it is also important to let the JVM warmup before the measuring starts. The JVM generates native code for the methods that are run at least a certain number of times, a number that can be specified by a compiler argument. This means that the performance will differ before and after the method has been compiled. To get fair performance tests it is therefore important to run the algorithm at least the specified number of times to make sure everything that should be compiled has in fact been compiled.

Another feature in Java is garbage collection. It allows the programmer to (almost) forget about memory management. This makes it easier for the programmer with the drawback of making performance non-deterministic since it is hard to predict when the garbage collection will occur. While doing performance tests it is desirable to avoid garbage collection as it pauses the program and therefore affects the measured execution time. With this in mind the tests are constructed in a way that no garbage collection is needed during the measuring. This gives a fair comparison between the versions.

5.4 VaR performance tests

Performance is measured on four different matrix sizes, see Table 5.2. The set of 256 scenarios represents roughly one year of historical data, counting only business days. The set of 1024 scenarios can be thought of as either four years of historical data or three years with one year covering particularly stressed market conditions; both being viable alternatives depending on the market. The choices of risk factor set sizes, 1 000 and 40 000, represents crude approximations of the numbers of risk factors necessary to perform historical VaR with instrument specific resolution for a small market and a large (or fragmented) market, respectively.
5.5. OPTIMIZING EACH VERSION

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Risk Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>1 024</td>
</tr>
<tr>
<td>256</td>
<td>39 936</td>
</tr>
<tr>
<td>1024</td>
<td>1 024</td>
</tr>
<tr>
<td>1024</td>
<td>39 936</td>
</tr>
</tbody>
</table>

Table 5.2: Matrix sizes used in the performance tests

As the execution time of a single calculation is too short to give good accuracy when measuring the time the tests are setup to run many calculations and then calculate the average execution time of a single calculation. Since the execution time is longer for bigger matrices the exact number of calculations also varies depending on the size of the matrix. This way the execution time gets long enough to give accurate measurements but short enough to perform all the necessary tests within a reasonable time. For the smallest matrix size the number of calculations run is 100 000 while for the biggest it is 1 000.

All tests are run several times to increase the statistic significance and protect against unpredicted disturbances to the tests, such as the operating system intervening in some way. The execution times presented in the report are always the median value of all the measured average execution times.

5.5 Optimizing each version

It is also important to find the optimal configuration for each implementation. You would obviously want to use the best one in a system and therefore it is also the one you want to test. Thus for each matrix the fastest serial version has to be found. For the tiled version the first step is to test which tile size gives best performance, then it is compared to the performance of the naive version.

For all OpenCL and Aparapi versions the optimal work group sizes have to be found. Since there are many possible configurations this is a time consuming task.

To find the optimal work group size of the OpenCL and Aparapi versions where every thread calculates a whole row there is only one variable to test, the work group size. There are two restrictions on the maximum size of the work group. It cannot exceed the number of work items (number of rows) nor the maximum work group size that the hardware can handle.

In the versions where the rows are split up into parts you can also vary how many parts the rows should be split into for each work group size. This gives more variants that have to be tested to find the optimal one. When the rows are split up and the work group calculates a single row, all versions with the word Split in the name (see Section 4.8) except GPULocalVectorsSplitBetter, there is also a restriction that the work group size cannot exceed the number of parts of each row. These tests could be automated into an auto-tuner that could be used to find the best configuration for the current hardware. Though due to the scope of this thesis
the auto-tuner is left as future research.

In both Java8 versions it is possible to specify the number of cores to use by specifying the number of threads to run. According to Amdahl’s law the speedup that can be achieved in a program is restricted by how large part of the program that can be parallelized and how many processors or cores it runs on. In practice however the speedup can stop increasing and actually start to decrease when adding more cores after a certain point. This is due to the arise of contention when each core gets too small calculations and therefore start to compete about getting new work. Because of this tests have to be run to find out the optimal number of cores for each of the tests in Table 5.2. In the tiled Java8 version it is also possible to specify the tile size.

Table 5.3 shows which serial and parallel Java implementation that performed best. For the problem sizes where the blocked algorithm was best the tile size is shown as the number of rows and columns. In Table 5.5 you can see the optimal work group sizes for all OpenCL and Aparapi versions. For the OpenCL and Aparapi versions with split rows the number of partitions per row can be seen in Table 5.4. Results on the optimal number of cores when running the parallel Java8 version on Bacall can be seen in Figure 5.1.

<table>
<thead>
<tr>
<th>Version</th>
<th>256x1024</th>
<th>1024x1024</th>
<th>256x39936</th>
<th>1024x39936</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java8</td>
<td>Naive</td>
<td>Naive</td>
<td>1</td>
<td>1024</td>
</tr>
<tr>
<td>Serial</td>
<td>Naive</td>
<td>Naive</td>
<td>4</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 5.3: Best serial and parallel Java version. Specifying the number of rows and columns in the two cases where the blocked algorithm performed best.

<table>
<thead>
<tr>
<th>Version</th>
<th>256x1024</th>
<th>1024x1024</th>
<th>256x39936</th>
<th>1024x39936</th>
</tr>
</thead>
<tbody>
<tr>
<td>AparapiCPUSplit</td>
<td>32</td>
<td>8</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>AparapiGPUSplit</td>
<td>32</td>
<td>4</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>AparapiJTPSplit</td>
<td>16</td>
<td>16</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>GPULocalSplit</td>
<td>16</td>
<td>16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GPULocalVectorsSplit</td>
<td>16</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GPUGlobalSplit</td>
<td>8</td>
<td>16</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>GPUGlobalVectorsSplit</td>
<td>4</td>
<td>8</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 5.4: Partitions per row
5.5. Optimizing Each Version

<table>
<thead>
<tr>
<th>Version</th>
<th>256x1024</th>
<th>1024x1024</th>
<th>256x39936</th>
<th>1024x39936</th>
</tr>
</thead>
<tbody>
<tr>
<td>AparapiCPU</td>
<td>16</td>
<td>1024</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>AparapiCPUSplit</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AparapiJTP</td>
<td>8</td>
<td>16</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>AparapiJTPSplit</td>
<td>16</td>
<td>16</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>AparapiGPU</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>AparapiGPUSplit</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>CPUOpenCL</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>GPUGlobal</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>GPUGlobalSplit</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>GPUGlobalVectorsSplit</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>GPULocalSplit</td>
<td>16</td>
<td>16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GPULocalVectors</td>
<td>4</td>
<td>32</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GPULocalVectorsSplit</td>
<td>16</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GPULocalVectorsSplitBetter</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>XeonPhi</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.5: Best workgroup sizes

Figure 5.1: Java8 Speedup per thread on Bacall
Chapter 6

Evaluation

In this chapter the ease of programming as well as the performance test results will be presented and evaluated.

6.1 Ease of programming

In this section the ease of programming will be discussed. As it is not possible to evaluate the ease of programming for the different versions in a truly objective way this section will be based on the author’s personal opinion. One measurement that might have been used is the complexity as the time it took to implement. As the knowledge increases for each successful implementation, it is hard to get a fair comparison. Also many improvements to old implementations were discovered along the way making it quite hard to give an accurate measure of the time needed for each implementation. Thus no such measures will be given.

6.1.1 OpenCL vs Aparapi

Aparapi is supposed to generate all the code required to offload a computation to the GPU. You write your kernel in Java and call it from Java. Then Aparapi generates the kernel code and the code to transfer data to and from the device. Even though the programmer can write the kernel in Java it is still a kernel to be executed on a GPU and has to be written as such. This means that the programmer still has to divide the calculation based on the id of the thread etc. At the moment Aparapi only supports primitive data types and no objects inside a kernel. Thus Aparapi has succeeded in simplifying GPU programming in some aspects though it still demands that you know the basics about GPU programming and the underlying hardware to get optimal performance.

Programming OpenCL demands you to both program the kernel, in OpenCL, and also write a host program, in another language such as Java. The host program copies the necessary input to the device and when the kernel is finished it copies
the result back to the host. This means more code to write than simply writing the whole program in the same language (using a reasonably high level language).

The biggest challenge with OpenCL on NVIDIA GPUs is that at the time of writing there are no debuggers to aid you. This makes the kernel a black box to which you send input and receive output from but gives you no indication as to what happens in-between. By storing extra information about the middle steps and adding them to the output you can get some hints about where the error occurs but this is not as convenient as a true debugger. Another way is to run the OpenCL kernel on another device for which there is a debugger and debug on that device.

The biggest difference between writing OpenCL and using Aparapi in terms of ease of programming is that you have to write code for the host to copy data to and from the device when writing OpenCL.

6.1.2 OpenCL: GPU vs CPU vs Xeon Phi

Here I group the CPU and coprocessor together as there is no real difference in how to program OpenCL kernels for these devices. Between the CPU/coprocessor and the GPU there are differences though. There is less to care about when programming for the CPU/coprocessor as these devices appear to get best performance with the simplest of the GPU kernels.

Since the CPU and coprocessor utilize caches you do not need to copy anything to local memory as everything will be cached automatically. No vectorization has to be implemented as the Intel SDK will do that for you. This leaves only the question of whether it is worth the effort to split the rows to increase the parallelism of the problem.

Since the Xeon Phi can run 228 threads, the CPU on Bacall can run 40 and the CPU on Valarauko 48 it will probably not improve performance to increase the parallelism since the smallest number of rows in the tests is 256. Splitting the rows when running on the Xeon Phi or the CPU made the algorithm return large errors in the result. Since there are no other large differences this is probably due to the automatic vectorization. Due to the limited scope of the thesis the investigation of why splitting rows on the Xeon Phi and CPU fails, if it is possible to implement a correct version and in that case if it improves performance is left for future research.

6.1.3 Java8

The Java8 solutions were the easiest to implement. It is very straightforward as everything is written in the same language and thanks to the parallel for-each stream there is very little code needed, as you can see in Figure 4.11.

6.2 Correctness

To evaluate the correctness of the implementations they were all compared to a serial reference implementation. The reference implementation was the naive algorithm
6.2. CORRECTNESS

(see Section 4.3) run on the CPU. When calculating with floating-points on different hardware the results usually differ a little due to how the floating-point calculations have been implemented in the hardware. Thus a small difference has to be expected and accounted for.

6.2.1 Accuracy

The versions running on the CPU with full rows gave exactly the same result as the serial version. This was not surprising as the same hardware should give the same results if the algorithm is the same. When running on the GPU with few risk factors the first 12 decimals were equal to the serial result. When increasing the risk factors the accuracy dropped by one decimal as you can see in Table 6.1. Splitting the rows decreased accuracy further when having many risk factors. This is likely because more partitions means more calculations that can give a small error. As all these errors are summed the final error is likely to be bigger the more partitions you have.

<table>
<thead>
<tr>
<th>Version</th>
<th>1024 Risk factors</th>
<th>39936 Risk factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>XeonPhi</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>GPUSplit</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>JavaVectorInstructions</td>
<td>12</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 6.1: Number of correct decimals with doubles

A calculation that is not accurate might not be valuable at all even though it runs very fast and a perfectly accurate calculation can be too slow to be useful. Thus it is relevant to measure both performance and accuracy. By doing that it is possible to choose the best solution depending on your specific needs.

When calculating VaR the final value is usually presented with two decimals. Thus all versions give good enough accuracy. If the value is supposed to be used in other calculations it might need more investigating how large the error can get in the end.

Accuracy when running with single-floating points were not as good as can be seen in Table 6.2. Here the errors are large enough to affect the first two decimals that are to be presented. This is clearly not desirable and therefore only a few performance tests were done with floats.

<table>
<thead>
<tr>
<th>Version</th>
<th>1024 Risk factors</th>
<th>39936 Risk factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>XeonPhi</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>GPUSplit</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2: Number of correct decimals with floats

51
6.3 Performance results

All versions have first been tested to find the optimal work group size. How this was done and which work groups were used can be seen in Section 5.5.

The Java8 versions runs with the maximum number of threads. Tests were done on Bacall to find the optimal number of threads for each matrix. Figure 5.1 shows the test result for these tests. Here we can see that the optimal performance is reached at about 20 threads for all matrix sizes except 1024 x 1024 which actually continues to perform better all the way up to the maximum of 40 threads. Since the maximum number of threads gave optimal performance (even if it can be achieved with less threads) all tests were performed with the maximum number of threads.

6.3.1 Performance with doubles

The performance tests show that the fastest solution in most tests was the parallel version written in Java8. Only on Sammy the GPU performed better than the CPU and only on the two biggest problem sizes. Using vector instructions gave only a small increase in performance, slightly larger on the small problem sizes than on the larger problem sizes. The achieved speedup on Valaruko have been split up into four figures, Figure 6.1-6.4, one for each problem size. Results for the tests on Sammy can be seen in Figure 6.5 and the results from Bacall can be seen in Figure 6.6. In the Appendix Test Data there are also figures showing the runtimes for each machine.

6.3.2 Performance with floats

As the accuracy was not good with floats there were only a few performance tests done with floats. One for the best GPU version on the biggest matrix which gave about 8% worse performance than Java8. The execution time of the GPU was 4317 microseconds compared to 3991 for Java8. The same test was performed with the Xeon Phi which ran slightly longer, 4526 microseconds, but at least managed to get 2 decimals correct as can be seen in Table 6.2.
6.3. PERFORMANCE RESULTS

Figure 6.1: Speedup Valarauko 256 x 1024

Figure 6.2: Speedup Valarauko 1024 x 1024
Figure 6.3: Speedup Valarauko 256 x 39936

Figure 6.4: Speedup Valarauko 1024 x 39936
6.3. PERFORMANCE RESULTS

Figure 6.5: Speedup on Sammy

Figure 6.6: Speedup on Bacall
6.4 Analysis

The fact that the parallel version on the CPU gives better performance than the GPU or the coprocessor implies that the VaR algorithm tested did not expose enough parallelism for the greater number of cores on these devices. The faster speed of the CPU then delivers better performance on these relatively few calculations. We can see in the Figures that performance on the GPU compared to the serial version increases with the number of scenarios and risk factors. If the algorithm were to run on a case with a larger number of scenarios or risk factors the GPU might give better performance. This is also supported by earlier research at KTH which shows that multiplication of two matrices gives a lot bigger speedup than the one achieved in this thesis with matrix-vector multiplication [25].

There is also a small overhead when calling a function written in another programming language like OpenCL, from within Java. As the execution times in the tests are short, especially when having few scenarios and risk factors, this overhead might be large enough to affect the result. When having larger problems with longer execution times the small overhead will not affect the results as much. In Section 6.4.4 this overhead is examined.

6.4.1 Winner of the performance tests

The tests shows that the CPU is the fastest processing unit when running VaR calculations from a system written in Java and using the tested number of scenarios and risk factors. Only on Sammy with a less powerful CPU the GPU managed to perform better on the two bigger problem sizes.

With unlimited resources and a server fully dedicated to VaR calculations the CPU would thus be the optimal processing unit. Running Java8 on Bacall with 1024 scenarios and 39936 risk factors the performance increases up till 20 threads and then stays quite steady at the same level up till the maximum of 40 threads. Therefore it is relevant to consider how many cores are free to use for the VaR calculations and how many cores are needed for other calculations. Already at 8-10 threads the CPU gets about the same result as the Xeon Phi on Valarauko, which delivers the second best performance on this problem size. Thus it is a tradeoff between performance and how many cores to leave available for other calculations.

Depending on the purpose of the machine and the resources available it could still be a better solution to leave the CPU for other calculations and have a GPU or coprocessor fully dedicated to the VaR calculations. This way the processor need only a single or a few cores for the VaR and can utilize the majority of its cores on other important calculations.

6.4.2 Performance of Aparapi

The results clearly show that writing kernels in Aparapi give poor performance. This can be seen in Figure 6.1-6.4 were all Aparapi versions running on the GPU
and CPU is slower than the serial version. The only version that actually increase performance is the one using a JavaThreadPool. As it is only the JavaThreadPool that increases performance there is no need to write an Aparapi kernel instead of writing the algorithm in Java with a JavaThreadPool, or parallel streams.

6.4.3 The optimal GPU version

In Figure 6.1 we can see that using local memory on the GPU gave decreased performance. This implies that the overhead of copying the series vector to local memory was bigger than the speedup achieved from the local memory. For the bigger problem sizes in Figure 6.2-6.4 we can see that the speedup from using local memory has become greater than the overhead as performance increases.

Splitting the rows did not improve performance in the case with 1024 risk factors, as can be seen in Figure 6.1-6.2. In these test we can also see that having the work groups calculating multiple rows is better than having them calculate a single row. When increasing the number of risk factors to 39936 splitting the rows does give better performance than calculating full rows which can be seen in Figure 6.3-6.4. In these two tests the version called GPULocalVectorsSplitBetter is the fastest. This implies that the problem size has become large enough to make all optimizations increase performance.

Which version to use depends on the size of the problem. Small problems do not need optimizations, they perform better without them. While calculating larger problem sizes needs more optimizations to get best performance from the GPU.

6.4.4 Overhead on GPU and coprocessor

Transfer of data

One advantage of running a pure Java version is that you do not get any overhead from copying data to and from the GPU or coprocessor. By copying the next vector in line while performing the calculation on the current vector it is possible to hide this overhead. To determine if the transfer time is the reason the Java8 implementation performed better than the GPU version tests were performed on Valarauko that measured the transfer time of the different vectors.

The larger vector of around 40000 elements took 56 µs to copy to the device. The whole calculation including transfer takes 3538 µs and 8578 µs respectively for 256 and 1024 scenarios. This means that the transfer times are around 1.6% and 0.7% respectively.

For the smaller vector on 1024 elements the transfer took 4 µs and the total calculation for the different number of scenarios took 245 with 256 scenarios and 407 µs with 1024 scenarios. This means that the transfer of the smaller vector is about 1.6% and 1% for 256 and 1024 scenarios respectively.

The transfer of the vector is at most 1.6% of the total execution time on the tested number of scenarios and risk factors. This shows that the transfer times are not a big slow down for the GPU and coprocessor in this case. In Table 6.3 we can
see that even if we remove the transfer times from the execution time the GPU will still not get better performance than Java8 in the performed tests.

<table>
<thead>
<tr>
<th></th>
<th>256 x 1024</th>
<th>1024 x 1024</th>
<th>256 x 39936</th>
<th>1024 x 39936</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>4 µs</td>
<td>4 µs</td>
<td>56 µs</td>
<td>56 µs</td>
</tr>
<tr>
<td>% of execution time</td>
<td>1.6%</td>
<td>1%</td>
<td>1.6%</td>
<td>0.7%</td>
</tr>
<tr>
<td>New execution time</td>
<td>241 µs</td>
<td>403 µs</td>
<td>3 482 µs</td>
<td>8 522 µs</td>
</tr>
<tr>
<td>Java8 execution time</td>
<td>35 µs</td>
<td>66 µs</td>
<td>990 µs</td>
<td>4 207 µs</td>
</tr>
</tbody>
</table>

Table 6.3: Overhead for transfer of data and the theoretical execution time without the overhead for the GPU on Valarauko

**Overhead of JNI calls**

Jogamps JOCL is using JNI to call the OpenCL kernel from within Java. This could be a disadvantage for the GPU and coprocessor. Therefore tests were done to see how much time is lost by using JNI to call on a function outside of Java. As the systems at Cinnober are written in Java this overhead is likely unavoidable, though it is still interesting to find out how large they are.

To test this two methods were implemented that performed a single addition, one in Java and one in OpenCL. The tests were setup as described in Chapter 5, with the OpenCL version running on the CPU like the Java version does. Results from running these performance tests on Valarauko show that the OpenCL version is 19 µs slower than the pure Java version. JNI thus appears to come with a 19 µs overhead. As this overhead is constant it will have a larger impact the smaller the problem size is which can be seen in Table 6.4.

<table>
<thead>
<tr>
<th></th>
<th>256 x 1024</th>
<th>1024 x 1024</th>
<th>256 x 39936</th>
<th>1024 x 39936</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>19 µs</td>
<td>19 µs</td>
<td>19 µs</td>
<td>19 µs</td>
</tr>
<tr>
<td>% of execution time</td>
<td>7.8%</td>
<td>4.7%</td>
<td>0.5%</td>
<td>0.2%</td>
</tr>
<tr>
<td>New execution time</td>
<td>226 µs</td>
<td>388 µs</td>
<td>3 519 µs</td>
<td>8 245 559 µs</td>
</tr>
<tr>
<td>Java8 execution time</td>
<td>35 µs</td>
<td>66 µs</td>
<td>990 µs</td>
<td>4 207 µs</td>
</tr>
</tbody>
</table>

Table 6.4: Overhead of JNI call and the theoretical execution time without the overhead for the GPU on Valarauko

**Total overhead**

Adding both these overheads together we get 75 µs overhead with 40000 risk factors and 23 µs overhead with 1024 risk factors. This corresponds to a maximum of 9.4% of the execution time on the smallest problem size. It would seem that this is a significant overhead though removing 23 µs from the execution time on the fastest GPU version still leaves the Java8 version 6 times as fast which can be seen in Table 6.5. All execution times including all overheads can be seen in Appendix III.
6.4. ANALYSIS

<table>
<thead>
<tr>
<th></th>
<th>256 x 1024</th>
<th>1024 x 1024</th>
<th>256 x 39936</th>
<th>1024 x 39936</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>23 µs</td>
<td>23 µs</td>
<td>75 µs</td>
<td>75 µs</td>
</tr>
<tr>
<td>% of execution time</td>
<td>9.4%</td>
<td>5.7%</td>
<td>2.1%</td>
<td>0.9%</td>
</tr>
<tr>
<td>New execution time</td>
<td>222 µs</td>
<td>384 µs</td>
<td>3463 µs</td>
<td>8503 µs</td>
</tr>
<tr>
<td>Java8 execution time</td>
<td>35 µs</td>
<td>66 µs</td>
<td>990 µs</td>
<td>4207 µs</td>
</tr>
</tbody>
</table>

Table 6.5: Total overhead and the theoretical execution time without the overhead for the GPU on Valarauko

6.4.5 Economical aspect

Another aspect on the choice of hardware is the cost, both the initial cost for building the system but also the daily cost to have it up and running. If the faster CPU is more expensive or has a lower performance per watt rating the GPU might still give more performance per dollar. The more a system is running the more it is possible to gain by using a processing unit with as high performance per watt rating as possible.

Since both Bacall and Valaruko hosts two CPUs that each need 130W [4, 24] this means they need 260W to run at full speed. The NVIDIA GPUs only need 225W running at full speed [26, 23] while the Xeon Phi needs 300W [27]. This has to be taken into account when building a production system that will be running daily. A quick look showed that the Kepler K20c GPU can be acquired for around 30000 SEK and the Tesla M2050 for around 25000 SEK. Buying two Xeon E5-2690v2 costs around 33000 SEK and two E5-2697v2 42000 SEK while the Xeon Phi can be acquired for around 12500 SEK. Thus the Xeon Phi appears to be cheapest to acquire while the GPUs appears to be cheapest to keep up and running.

Looking at the tests gives another picture though. In the case with 1024 scenarios and 39936 risk factors the CPU on Bacall needs 6 threads to perform almost equally with the Tesla M2050 GPU. If the wattage is distributed equally to all cores this would mean the CPU needs around 39W. By splitting the rows the number of work items should be high enough for the GPU to be fully utilized, though it is harder to prove how many of the cores on the GPU is actually running. Getting a performance per watt ratio that is lower than the CPU would mean that the M2050 only can run a sixth of all its cores and the Xeon Phi can run 30 of its 228 threads. As the number of work items is high this seem unlikely and the CPU should therefore have a better performance per watt rating in this case.

If we again look at the case with 1024 scenarios and 39936 risk factors where the CPU has lowest advantage over the GPU we can see that the best CPU (Xeon E5-2690v2) gives 84% better performance than the best GPU (M2050) while the price is only about 32% higher. Since this was the worst case for the CPU, it has a higher performance per dollar ratio in all tested cases. Looking at the Xeon Phi it has about 40% of the CPU performance but at 164% lower price.

There is another part to the economical aspect, perhaps an even more important aspect, and that is the cost of the implementation and the maintenance of the code.
CHAPTER 6. EVALUATION

Less time spent on implementing a program means less expenses for a company. As stated in Section 6.1.3 the Java8 version was the easiest to implement. It is also the one consisting of the least amount of code that needs to be maintained. Another advantage is that there is no need to know OpenCL or the GPU programming model to maintain the Java code. At a company like Cinnober with a lot of Java experts this is of course positive.

The CPU has highest performance per watt and second best performance per dollar after the Xeon Phi. If a system is supposed to run for a long time the performance per watt is more interesting than the performance per dollar. Developing and maintain the code is likely an even larger cost than the hardware cost. Therefore the Java8 implementation is the winner from an economical aspect as it is easiest to develop and maintain and runs on the CPU.

6.5 The verdict

There is no optimal hardware that is the best in all cases. The CPU is a clear winner when it comes to performance on the VaR algorithm with the tested number of scenarios and risk factors but there are more factors that have to be taken into consideration when designing a high performance system. Since Cinnober is mainly focused on Java solutions the simplicity with using Java all the way is a heavy argument that together with the performance results can be argued to be the best solution in this specific case.

6.6 Future research

There are still some interesting areas that are left unexplored due to the scope of the thesis. In future research it would be interesting to investigate if the Intel CPUs and coprocessor can achieve better performance by writing the vectorization yourself instead of relying on Intels automatic vectorization. When doing this it would also be interesting to see if it is possible to get correct results when splitting rows and if this increases performance. It would also be interesting to find out if writing and running a C program instead of OpenCL on the coprocessor and CPU increases performance. Since it theoretically should be possible to run a Java program on the coprocessor this is also an interesting area to look into. Though at the time of writing there is no official support for this.

The auto-tuner mentioned in Section 5.5 would also be an interesting area to investigate. As the approach to find the optimal work group size is very straightforward, simply test the possible variants and choose the best, this should be easy to implement. The tuning will be time consuming if it is not possible to find a way to theoretically limit the number of work groups to test. If this is to be done it has to be in a clever way so that the optimal version will be tested and found but the worst versions can be excluded. If GPUs are to be used an auto-tuner like this would be very valuable for testing new hardware.
Part III

Bibliography
Bibliography


Appendices
Glossary

Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>The amount of work that can be done per time unit.</td>
</tr>
<tr>
<td>Efficiency</td>
<td>A measure for how well cores in a parallel version are utilized.</td>
</tr>
<tr>
<td>Speedup</td>
<td>The number of times faster a parallel version is compared to the serial version of the same program.</td>
</tr>
<tr>
<td>Profit and Loss</td>
<td>The resulting vector in a VaR calculation. Contains the estimated profit or loss for each scenario.</td>
</tr>
</tbody>
</table>

Table 1: Glossary

Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full word</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Graphics Processing Unit</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>VaR</td>
<td>Value at Risk</td>
</tr>
<tr>
<td>SSE</td>
<td>Streaming SIMD Extensions</td>
</tr>
<tr>
<td>AVX</td>
<td>Advanced Vector Extensions</td>
</tr>
<tr>
<td>PCIe</td>
<td>PCI Express</td>
</tr>
<tr>
<td>P&amp;L</td>
<td>Profit and Loss</td>
</tr>
</tbody>
</table>

Table 2: Abbreviations
Test Data

Figure 1: Runtimes Valarauko 256 x 1024
Figure 2: Runtimes Valarauko 1024 x 1024

Figure 3: Runtimes Valarauko 256 x 39936
Figure 4: Runtimes Valarauko 1024 x 39936

Figure 5: Runtimes Bacall
Figure 6: Runtimes Sammy 1024 risk factors

Figure 7: Runtimes Sammy 39936 risk factors
Figure 8: Runtimes per thread on Bacall 256 x 1024

Figure 9: Runtimes per thread on Bacall 1024 x 1024
Figure 10: Runtimes per thread on Bacall 256 x 39936

Figure 11: Runtimes per thread on Bacall 1024 x 39936