Tuning and Performance Analysis of the Black-Scholes Model

Robin Hansma

June 9, 2017

Supervisor(s): prof. dr. R.V. (Rob) van Nieuwpoort, A. Sclocco
Signed: prof. dr. R.V. (Rob) van Nieuwpoort, R. Hansma
Abstract

There are different options to increase performance without consuming more power, such as using more power efficient hardware, or improving the efficiency of applications. A way to improve the efficiency of an application is to use auto-tuning to find the best configuration for its parameters. In this thesis we explain the performance of the Black-Scholes kernel on both GPUs and CPUs. We conclude that auto-tuning an already optimised kernel still makes sense, the performance increase for the GPUs is in the range of 8-11% and for the CPU even 44% (the original kernel was optimised for GPUs).
# Contents

1 Introduction .............................................. 7
  1.1 Research Question .................................. 7
  1.2 Thesis outline ....................................... 8

2 Background .............................................. 9
  2.1 Auto-Tuning .......................................... 9
  2.2 OpenCL .............................................. 9
  2.3 Black-Scholes ....................................... 10

3 Related Work .......................................... 11
  3.1 Accelerating Radio Astronomy with Auto-Tuning ...... 11
  3.2 Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures .............................................. 11

4 Implementation ......................................... 13
  4.1 Black-Scholes ....................................... 13
    4.1.1 Optimisations ................................... 13

5 Experiments ............................................ 15
  5.1 Experimental Setup .................................. 15
  5.2 Performance Results ................................ 15
    5.2.1 Black-Scholes ................................... 16

6 Future work ........................................... 29
  6.1 Generalise findings .................................. 29

7 Conclusion ............................................... 31
In the past, it was common to increase the performance of supercomputers by simply adding more cores, either in the form of more nodes, multi-core CPUs or recently many-core GPUs. In order to scale to exascale, extra attention should be payed to the efficiency of the supercomputers and the efficiency of the algorithms running on them. The recent change from single core processors to multi-core processors required a shift in mindset of developers [7] and required a lot of programs to be rewritten to make the most out of the new architectures. Rewriting programs requires algorithm-specific knowledge and an upfront investment without any knowledge on how performant the new implementation will be. The result is that a lot of scientific [13] and commercial programs are still not optimised for multi-core processing and thus waste processing power.

The recent change of interest in efficiency of supercomputers as opposed to raw performance has led to the usage of a new set of benchmarks to measure this. The results of this set of benchmarks are summarised in the Green500 list, which lists the top 500 supercomputers based on efficiency (MFLOPS/W). The top 10 of this list contains five different architectures and seven different main accelerators as shown in table 1.1. This illustrates the challenge developers face every day: which platform is best to optimise the algorithm for? To make this decision even harder, these architectures change from year to year.

Auto-tuning frameworks can be used to improve performance of algorithms for a specific architecture. This improves the performance of the algorithm itself, but also improves the portability of performance [13]. Auto-tuning automatically searches the predefined configuration space for the best configuration on a architecture, this way a developer doesn’t need to know the hardware specifics to develop an efficient implementation of the algorithm.

1.1 Research Question

The focus of this thesis is to explain why certain configurations are more efficient on certain architectures than on others. Answering this question can provide an useful insight into architectures and help make them better performing. It also has great use for the auto-tuning field itself, because knowing in advance which configurations are more likely to perform well, can decrease the configuration space and thus the time required to find the optimal configuration.

The research question of this thesis is therefore: "Why some kernel configurations are more efficient on certain architectures and not on others. In particular, what we want to find out is the relationship between the configuration and performance of a kernel on different architectures.”

This thesis is based on the work of Alessio Sclocco [13]; an analysis of the relationship between this thesis and [13] will be presented in section 3.1.

We will extend TuneBench, the auto-tuning framework developed by Alessio Sclocco, by adding another tunable kernel to it.
**Supercomputer full specifications**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Supercomputer</th>
<th>Specifications</th>
<th>Accelerator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NVIDIA DGX-1, Xeon E5-2698v4 20C 2.2GHz, Infiniband EDR, NVIDIA Tesla P100</td>
<td>NVIDIA Tesla P100</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100</td>
<td>NVIDIA Tesla P100</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ZettaScaler-1.6, Xeon E5-2618Lv3 8C 2.3GHz, Infiniband FDR, PEZY-SCnp</td>
<td>PEZY-SCnp</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway</td>
<td>Sunway SW26010</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PRIMERGY CX1640 M1, Intel Xeon Phi 7210 64C 1.3GHz, Intel Omni-Path</td>
<td>Intel Xeon Phi 7260</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path</td>
<td>Intel Xeon Phi 7250</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Cray XC40, Intel Xeon Phi 7230 64C 1.3GHz, Aries interconnect</td>
<td>Intel Xeon Phi 7230</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Cray CS-Storm, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, Nvidia K80</td>
<td>NVIDIA K80</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect</td>
<td>Intel Xeon Phi 7250</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>KOI Cluster, Intel Xeon Phi 7230 64C 1.3GHz, Intel Omni-Path</td>
<td>Intel Xeon Phi 7230</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: The top 10 of the Green500 of November 2016 with five different architectures and seven different main accelerators

### 1.2 Thesis outline

The second chapter provides some background information required to understand the rest of the thesis; in particular the auto-tuning framework, the implemented kernel and the language in which the kernel is implemented are discussed. In the third chapter we present a selection of papers that are important for this thesis, and in particular an overview of [13].

The implementation and the optimisations of the kernel implemented by myself are discussed in the fourth chapter. The fifth chapter discusses the experimental setup, the experiments and the results. Chapter six proposes some topics for future research in this research area. In the last chapter the conclusion of the thesis is presented.
In this chapter we’re going to introduce all background information that is necessary to understand the rest of this thesis. The following three sections describe how auto-tuning works, what OpenCL is and why it's used and some background information on the kernel. A kernel, as used in this thesis, is a small program which performs only one task. These kernels are as closely related to real world problems as possible, but should generate reproducible results. So the outcome could be compared to a sequential implementation to make sure the results are correct.

2.1 Auto-Tuning

The process of auto-tuning consists of automatically running a kernel with several different configurations to find the best configuration possible. A configuration is a combination of parameters, for example the number of threads and the number of times a loop is unrolled. The auto-tuning framework tests all possible configurations in a predefined configuration space. A possible configuration space of the number of threads could be 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.

If more than one parameter is tunable, the optimisation space grows as the Cartesian product of the values of each parameter. This growth is rapid and represents one of the main challenges of auto-tuning.

2.2 OpenCL

TuneBench uses OpenCL, which stands for Open Computing Language, as language for the kernels. This language is chosen because of the possibility to compile the OpenCL code to native code for GPUs, CPUs and hardware accelerators. Some supported hardware manufacturers are Intel, AMD, NVIDIA, IBM and ARM, for a complete list see the OpenCL website by Khronos [3], the maintainer of OpenCL. The OpenCL code is compiled at run-time, this ensures the code could be run by any supported platform without recompiling the kernel manually.

A disadvantage of trying to write universal code is that it’s not always the most efficient code possible. In order to make the most out of an architecture, architecture-specific code should be written. This however decreases the portability of the performance and requires architecture-specific knowledge. However, most of the time, the performance of a universal OpenCL implementation is close to a native implementation [13, 9].

OpenCL has to abstract away the exact details of the underlying platform in order to provide portability. Therefore it introduces concepts like work-groups and work-items. A work-group consists of one or more work-items which all are executed concurrently within a single compute unit [14]. A compute unit may be a single core, a SIMD unit, or any other element in the OpenCL device capable of executing code. Each work-item in a work-group will execute concurrently on a single compute unit [14], thus work-items are comparable to threads but not exactly the same. Its up to the implementation how the work-items are scheduled and if its actually treated as a
thread. Within a work-group, local memory is available, this local memory is shared within the work-group and is faster than the global memory [14].

2.3 Black-Scholes

In this thesis we are going to implement the Black-Scholes algorithm. The Black-Scholes model is a model which estimates the cost of options on the European financial markets. It’s not required to understand how this model works exactly in order to follow the rest of this thesis but some background does help explaining the performance of the kernel.

The model consist of two types of options, the call option and the put option. An option is a security giving the right to buy (call) or sell (put) an asset, subject to certain conditions, within a specified period of time [6]. The distinction with an American option is that European options can only be exercised on a specified future date, while American options can be exercised up to the date the option expires.

To be able to estimate the future price of the call and put options a couple of variables are required. These variables are the current stock price, the strike price of the options, the duration of the options in years, the riskless rate of return and the stock volatility. The riskless rate of return is the interest rate without calculating risk in to the equation. This is an assumption made by the Black-Scholes model and of course doesn’t hold in the real world. The stock volatility indicates how much the stock price changes.

The BlackScholes kernel used in this thesis is based on code developed by NVIDIA [2]. By using this kernel we focus on tunable optimisations instead of the implementation of the kernel itself.
CHAPTER 3

Related Work

In this chapter some related papers are discussed on which this thesis is based. The most important being the thesis of Alessio Sclocco [13], of which the framework TuneBench is extended. The second paper is an important paper which discusses the optimising of stencil computations. By extensively discussing the optimisations per architecture an insight in the performance of the different architectures is given.

3.1 Accelerating Radio Astronomy with Auto-Tuning

As mentioned earlier, this thesis is based on the work described in [13]. This research mainly focused on the radio astronomy and how the applications used in that field could be optimised. The possible techniques explored are using many-cores and auto-tuning. Also the question how difficult auto-tuning is, is explored. To be able to answer whether auto-tuning provides a possible solution, the framework TuneBench was developed which contains five kernels.

The kernels are used to run on several different platforms, CPUs, GPUs and accelerators. The platforms used are AMD Opteron 6172, AMD HD6970, AMD HD7970, AMD FirePro W9100, AMD R9 Fury X, Intel Xeon E5620, Intel Xeon E5-2620, Intel Xeon Phi 5110P, Intel Xeon Phi 31SIP, NVIDIA GTX 580, NVIDIA GTX 680, NVIDIA K20, NVIDIA GTX Titan, NVIDIA K20X, NVIDIA GTX Titan X, NVIDIA GTX 1080. By using TuneBench to get insights in to the optimisation space, the difficulty of auto-tuning could be studied.

The conclusions drawn from examining the optimisation spaces are that completely memory-bound applications are easier to tune than applications that, by exposing date-reuse through tunable parameters can be made almost compute-bound. The difficulty of tuning can also be a function of the input size. Another conclusion that can be drawn is that tuning many-core accelerators is, in general, difficult, but application-specific knowledge can help prune the search space of tunable parameters.

The last conclusion is that there is less correlation between an application being memory- or compute-bound and it having a more or less portable optimum configuration. The evidence found in [13] shows that the optimum is not really portable among different platforms, not even for the same input size. However some parameters are stable and do not vary at all. The variability of optimal configurations seem to be increasing in newer architectures.

3.2 Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures

After years of simply increasing the frequency and other optimisations to increase the per core performance, the performance is now mainly increased by adding more cores. This presents a lot of new challenges and thus several architectural approaches. It’s not yet clear what architectural
philosophy is best suited for what class of algorithms. This makes optimising for a new architecture an expensive task, because it’s not clear in advance whether it will perform better than the current architecture. Auto-tuning helps solving this problem by automatically finding the best configuration for several different architectures, which makes an algorithm extremely portable.

This paper uses stencil operations as benchmark for several architectures. These kernels can be parallelised very well and have a low computational intensity, offering a mixture of opportunities for on-chip parallelism and challenges for associated memory systems (and is thus memory bound).

The used architectures are the Intel Xeon E5355, AMD Opteron 2356, Sun UltraSparc T2+, IBM QS22 PowerXCell 8i Blade and the NVIDIA GTX280. An important thing to note is that the Geforce GTX280 has a notable faster onboard memory capacity of 1GB, when a problem size of bigger than 1GB must be handled, the GPU should switch to the slow DRAM over the even slower PCIe bus, which greatly reduces the throughput and thus performance.

The comparison of the architectures shows that the applicable optimisations and the effect of them is highly dependent on the architecture. There are off course some optimisations only available for certain architecture, for example SIMD is only available when implemented (which only is the case for Intel). On the other hand some optimisations are very effective on some architectures, like increasing the number of threads on the Geforce GTX280, while much less effective on the CPU’s [8].
CHAPTER 4

Implementation

This chapter discusses the implementation of the Black-Scholes kernel and the optimisations applied.

4.1 Black-Scholes

The original implementation of the Black-Scholes kernel follows the Black-Scholes model [6] and is discussed by NVIDIA [11], some important parts are highlighted here. In the body of the kernel the future price of one option at the time is calculated. To calculate the value of the N options, the kernel is executed N times with a different input option. The cumulative normal distribution function is not implemented in C++ so a rational approximation is used. All calculations are done using single precision floats.

4.1.1 Optimisations

The tunable parameters that we added are: the number of threads, the number of loop unrolls and vectorisation with different vector sizes. The original code does not use loop unrolling, nor vectorisation. The number of threads is changed by setting the dimensions of the work group as discussed in section 2.2.

Loop unrolling is a process in which the body of the loop is repeated multiple times, while updating the control logic of the loop. An example can be seen in figure 4.1, where the original loop is unrolled with an unroll factor of 3. By unrolling a loop the program size increases in an attempt to decrease the execution time. The improved execution time can be achieved by an increase in instruction-level parallelism, registry locality and memory locality [10, 12]. For GPUs this optimisation comes with a trade-off, when increasing the number of registers per thread, the number of threads that can execute concurrently decreases [10].

The last optimisations applied to this kernel is the vectorisation technique. This technique requires special vector processors and vector registers to profit of the optimisation. Intel architectures support vectorisation by using SIMD (Single Instruction Multiple Data) instruction sets like MMX, SSE and AVX. SIMD instructions run the same instruction on multiple data elements at the same time, for example when multiplying an array of 4 elements by 2 as shown in figure 4.2. A schematic drawing of a SIMD instruction in a processor is shown in figure 4.3. Modern GPUs don’t support vectorisation, this is because instructions on the GPU are scheduled in such a manner that memory latency is hidden as much is possible by default, by using light-weight threads instead of vectors.

The OpenCL compiler only supports vectors of size 2, 3, 4, 8 or 16. So only the configurations where the loop unroll factor is equal to one of these values is computed. The rest of the configurations are skipped because loop unrolling and vectorisation are applied at the same time.
# Before loop unrolling:
for (int i = 0; i < 6; i++) {
    print i;
}

# After loop unrolling:
for (int i = 0; i < 6; i+= 3) {
    print i;
    print i + 1;
    print i + 2;
}

Figure 4.1: An example of loop unrolling, the above for loop is unrolled to three separate statements

# Before vectorisation:
a[0] = a[0] * 2
a[1] = a[1] * 2

# After vectorisation:
a *= 2

Figure 4.2: Vectorisation makes it possible to apply one instruction to multiple data elements at the same time, the four statements on top are equal to the one below when vectorisation is enabled

![Figure 4.3: A schematic drawing of a SIMD instruction in a processor, source Wikipedia](image)
CHAPTER 5

Experiments

After discussing the experimental setup in the first section of this chapter, we later discuss the results of the experiments. Each experiment is executed on all of the devices discussed in the experimental setup. We try to provide an explanation for the observed behaviour of the experiments by holding the results next to the hardware specifications.

5.1 Experimental Setup

All experiments are run on the DAS-5 supercomputer. We've used the VU cluster and the devices used for running the experiments are listed in table 5.1. The five devices are based on four different architectures, this way it's possible to make a comparison between the different architectures. The exact details of the GPUs and accelerators can be found in the appendix.

The Tesla K20 and K40 are specifically designed for HPC purposes, while the Titan X (Maxwell) and Titan X (Pascal) in the first place are designed for gaming. Therefore the K20 and K40 have ECC memory to correct memory errors and also have a higher double precision performance. The ECC bits are stored in the main memory which reduces the size left for application usage by 10% for the K20 and 6.25% for the K40 accelerator. The experimental setup has ECC memory disabled for both the K20 and K40 and thus have access to all of the installed memory.

There is a configurable boost option available for both the K20 and K40. This boost option increases the clock speed of the shaders to temporarily improve performance, and is only available when there is enough power headroom (maximised at 235W). During the experiments the default clock speeds of 705MHz and 745MHz respectively are used. All experiments are repeated 1000 times, the average value of the experiments is used in this thesis.

<table>
<thead>
<tr>
<th>Device</th>
<th>Architecture</th>
<th>OpenCL Version</th>
<th>GFLOP/s</th>
<th>GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GTX TitanX</td>
<td>Maxwell</td>
<td>OpenCL-NVIDIA 8.0</td>
<td>6144</td>
<td>336</td>
</tr>
<tr>
<td>NVIDIA GTX TitanX</td>
<td>Pascal</td>
<td>OpenCL-NVIDIA 8.0</td>
<td>10157</td>
<td>480</td>
</tr>
<tr>
<td>NVIDIA Tesla K20</td>
<td>Kepler</td>
<td>OpenCL-NVIDIA 8.0</td>
<td>3524</td>
<td>208</td>
</tr>
<tr>
<td>NVIDIA Tesla K40</td>
<td>Kepler</td>
<td>OpenCL-NVIDIA 8.0</td>
<td>4291</td>
<td>288</td>
</tr>
<tr>
<td>Intel E5-2630</td>
<td>Sandy Bridge</td>
<td>OpenCL-Intel 4.5-mic</td>
<td>307.2</td>
<td>42.6</td>
</tr>
</tbody>
</table>

Table 5.1: The experimental setup with one Intel CPU and four NVIDIA GPUs and accelerators, the theoretical maximum performance is for single precision floats

5.2 Performance Results

In this section we discuss the performance results of the Black-Scholes kernel. The plots shown are a histogram of the optimisation space, which shows the performance on the y-axis and the
input size on the x-axis. The thicker the bar is, the more configurations are found with a certain performance. In other words, each vertical plot is a histogram. The other plots are plots of the performance per input size, where the x-axis shows the number of threads and the y-axis the performance in GFLOP/s. Not all plots are shown in this section, only the plots of input size 4,000, 40,000 and 2,560,000 options are shown below. The other plots can be found in the appendix. The maximum input size used is 2,560,000 options, this value is chosen because this value was the maximum value which could complete all tune runs, a higher input size would give an out of resources error when tuning with vector size 16 and 1024 threads. The true maximum performance is thus not yet reached with this input size. In this chapter, when we refer to a vector size of 0, we mean that there is no vectorisation of the code.

5.2.1 Black-Scholes

The Black-Scholes kernel was first tuned on the NVIDIA Titan X (Maxwell), a subset of the results is shown in figures 5.1 - 5.4. Figure 5.1 shows the minimum and maximum floating point operations per input size and also the distribution of the configurations.

![Figure 5.1: The configuration space of the BlackScholes kernel in GFLOP/s when tuning on a Titan X (Maxwell) GPU using varying input sizes](image)

When looking at the histogram its clear that the improvement of the performance of the kernel is very dependent on the input size. With a low input size of 4000 options the maximum and minimum are very close to each other. The input size is too low to fully utilise the hardware. When increasing the input size the performance increases significantly. The 75th percentile of input sizes till 64,000 is closer to the median, than to the maximum. This means that it’s relatively hard to get the maximum performance out of this architecture, since most configurations have median performance or less.

In figure 5.2 the plot using an input size of 4000 is shown. This plot supports the observation made earlier using figure 5.1, the performance doesn’t vary much and is mainly limited by the input size. Increasing the input size to 4000 increases the peak performance to 140 GFLOP/s as can be seen in figure 5.3. The differences between the configurations is now clearly visible. When using a low number of threads, a bigger vector size has a higher performance. The vector size of 8 is at its peak performance when using 4 threads and degrades at the increase of the number of threads. The configurations using vectors of size 0 or 2 are performing the best and
the peak performance is reached at 32 to 128 threads. After 128 threads the performance drops till the performance is equal to the performance of vector sizes 3 and 4.

Figure 5.2: The performance of the BlackScholes kernel on the Titan X (Maxwell) using an input size of 4000 options with a varying number of threads and vector size

Figure 5.3: The performance of the BlackScholes kernel on the Titan X (Maxwell) using an input size of 40000 options with a varying number of threads and vector size

What we can conclude from the analysis of the plot is that increasing the amount of compu-
tation per thread (increasing the vector size) doesn’t increase the performance. The theoretical peak performance hasn’t been reached yet, so this means that the registers are full and therefore the relatively slow main memory has to be used for storing and retrieving data. An important observation to make is that the Titan-X doesn’t have a special vector compute-unit available. The increase in the number of threads does improve the performance, because of the increased parallelism. Increasing the parallelism does also increase the register usage for a multiprocessor. So after all the registers are used, increasing the parallelism hurts the performance.

Theoretical peak performance hasn’t been reached yet, so this means that the registers are full and therefore the relatively slow main memory has to be used for storing and retrieving data. An important observation to make is that the Titan-X doesn’t have a special vector compute-unit available. The increase in the number of threads does improve the performance, because of the increased parallelism. Increasing the parallelism does also increase the register usage for a multiprocessor. So after all the registers are used, increasing the parallelism hurts the performance.

Figure 5.4: The performance of the BlackScholes kernel on the Titan X (Maxwell) using an input size of 2560000 options with a varying number of threads and vector size

The other input sizes show roughly the same behaviour, the base and peak performance are at 111 GFLOP/s and 691 GFLOP/s respectively. When using a low number of threads the configurations with a higher vector size are more efficient because there are less work-groups and thus less context-switches between them. A context-switch normally does add little overhead on a GPU (since all scheduling is done in hardware), but in this case the number of work-groups is at its maximum and thus the combined overhead is significant. Also when choosing a configuration with few threads the GPU isn’t fully utilized, because NVIDIA works in lock-step warps of 32 threads.

In figure 5.5 the histogram of the optimisation space of the Titan X (Pascal) is shown. This plot follows the same pattern as the plot in figure 5.1. The peak performance is a bit higher at 918 GFLOP/s, an increase of roughly 30%, which is less of an increase than the increase in theoretical GFLOP/s which is roughly 65%. What does stand out is that the 75th percentile and median are higher for the Titan X (Pascal) than for the Titan X (Maxwell), which means that there are relatively more configurations closer to the maximum.

When we look at figures 5.6 to 5.8 we see that the behaviour is the same as for the Titan X (Maxwell). The most notable change is found in the plot for input size 2.56 million, figure 5.8, where the vector sizes 0, 2, 3 and 4 are all within 50 GFLOP/s. This is considerably closer to each other than in figure 5.4, where the difference between those vector sizes is almost 200 GFLOP/s and supports the observation made earlier, that the Titan X (Pascal) is easier to tune than the Titan X (Maxwell). An explanation for this behaviour hasn’t been found. That the Titan X (Maxwell) and Titan X (Pascal) behave mostly the same is because the architecture design of the latter is almost identical to the former.

The plot in figure 5.9 shows the optimisation space of the K20 accelerator. In contrast to
Figure 5.5: The number of floating point operations of the BlackScholes kernel when tuning on a Titan X (Pascal) GPU using varying input sizes.

Figure 5.6: The performance of the BlackScholes kernel on the Titan X (Pascal) using an input size of 4000 options with a varying number of threads and vector size.

the earlier observed Titan X (Pascal) the median and 75th percentile are closer to the minimum. This suggests that this accelerator is harder to tune than the Titan X.

Figures 5.10 to 5.12 show the performance per number of threads per vector size for the K20 accelerator. The performance of the K20 benefits significantly from an increase in threads as can
Figure 5.7: The performance of the BlackScholes kernel on the Titan X (Pascal) using an input size of 40000 options with a varying number of threads and vector size.

Figure 5.8: The performance of the BlackScholes kernel on the Titan X (Pascal) using an input size of 2560000 options with a varying number of threads and vector size.

This can be explained by the configuration of the CUDA cores. There are more CUDA cores per multiprocessor for the K20 than for the Titan-X (Maxwell), namely 192 against 128. However there is a clear drop in performance visible if more than 256 threads are used. This is caused by the fact that there aren’t enough registers or multiprocessors...
Figure 5.9: The number of floating point operations of the BlackScholes kernel when tuning on a K20 accelerator using varying input sizes available to process the threads. The number of registers of both devices is the same, but the K20 has 13 multiprocessors and the Titan-X (Maxwell) has 24 multiprocessors.

As the K40 is a faster version of the K20, the behaviour of the K40 is almost identical to the K20. The optimisation space, shown in figure 5.13, is the same but with a higher performance.
Figure 5.11: The performance of the BlackScholes kernel on the K20 using an input size of 40000 options with a varying number of threads and vector size.

Figure 5.12: The performance of the BlackScholes kernel on the K20 using an input size of 2560000 options with a varying number of threads and vector size.

Again noteworthy that the K40 is harder to tune than the two Titan-X devices. The K40 also has a drop in performance when using a higher number of threads, but this drop is after 512 threads instead of 256 threads. This can be explained when looking at the specification of the K40, this accelerator has 15 multiprocessors instead of 13 multiprocessors for
Figure 5.13: The number of floating point operations of the BlackScholes kernel when tuning on a K40 accelerator using varying input sizes.

The last architecture tested is the Intel Xeon E5-2630 processor. The optimisation space of this device can be seen in figure 5.17. Most of the configurations are close to the median performance of the kernel, this shows that the kernel is not easily tuned for maximum performance.
but can get median performance fairly easy. When using an input size of 2,56 million options the performance drops. We’ve investigated this and concluded that the data must be too big to fit in the L3-Cache, the size of the L3-cache being 20MB and the total size of the input and output array being 20,48MB. This causes the processor to retrieve the data partially from the
memory which is much slower than the cache.

![Graph showing floating point operations (GFLOP/s) for varying input sizes](image)

**Figure 5.17:** The number of floating point operations of the BlackScholes kernel when tuning on a Intel E5-2630 CPU using varying input sizes

The figures 5.18 and 5.19 show that using a relatively small input size an increase in number of threads negatively effects the performance for vector size 2 and bigger. For vector size 0 the performance stays the same when increasing the number of threads. It makes sense that an increase in threads above the 128 threads don’t, or negatively, effect the performance since the number of threads for a processor like the Xeon E5-2630 is relatively small compared to a GPU. The number of threads available for the Xeon E5-2630 is 16.

The plot in figure 5.20 shows that vector sizes bigger than 0 provide a better performance. The best performance is when using a vector size of 4 or 8, this is because of the AVX instruction set which is supported by the processor. This instruction set supports 8 single precision floats at the time.

To provide a baseline we’ve executed the original kernel on all devices with an input size of 1.28 million. This way we can see whether the tuning has effect, even on such an already optimised kernel as the Black-Scholes kernel. The results can be found in table 5.2. We can conclude from the results that the performance increase is significant, namely 8 to 44%. The biggest performance increase can be seen when tuning the Intel E5-2630, which is explained by the fact that the original kernel is developed for NVIDIA GPUs, the default settings are thus quite inefficient for a CPU.

<table>
<thead>
<tr>
<th>Device</th>
<th>GFLOP/s (Original)</th>
<th>GFLOP/s (Tuned)</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GTX TitanX (Maxwell)</td>
<td>601.94</td>
<td>658.61</td>
<td>9.41%</td>
</tr>
<tr>
<td>NVIDIA GTX TitanX (Pascal)</td>
<td>777.02</td>
<td>864.56</td>
<td>11.27%</td>
</tr>
<tr>
<td>NVIDIA Tesla K20</td>
<td>382.90</td>
<td>415.03</td>
<td>8.39%</td>
</tr>
<tr>
<td>NVIDIA Tesla K40</td>
<td>459.25</td>
<td>500.49</td>
<td>8.98%</td>
</tr>
<tr>
<td>Intel E5-2630</td>
<td>80.91</td>
<td>116.52</td>
<td>44.01%</td>
</tr>
</tbody>
</table>

**Table 5.2:** The maximum performance of the original implementation of the kernel compared with the maximum performance of the tuned version of the kernel using input size 1.280.000 options
Figure 5.18: The performance of the BlackScholes kernel on the E5-2630 using an input size of 4000 options with a varying number of threads and vector size.

Figure 5.19: The performance of the BlackScholes kernel on the E5-2630 using an input size of 40000 options with a varying number of threads and vector size.
Figure 5.20: The performance of the BlackScholes kernel on the E5-2630 using an input size of 2560000 options with a varying number of threads and vector size.
We’ve made a first effort to explain the performance differences between architectures. In order to be able to fully understand the performance differences further research should be done. This chapter describes some of the possible research directions and some possible extensions for TuneBench.

6.1 Generalise findings

In order to completely understand what aspects of an architecture make an application better performing, the performance of more kernels should be examined. Also more devices should be tested so we can compare AMD architectures with NVIDIA architectures for example, or the Xeon Phi which has in terms of architecture a completely different design.
The goal of this thesis was to explain "why some kernel configurations are more efficient on certain architectures than on others". With help of the TuneBench framework we've examined the Black-Scholes kernel on the NVIDIA Maxwell, Pascal and Kepler architecture, and Intel Sandy Bridge architecture. A couple of conclusions can be drawn from this research.

The first, and most obvious, conclusion is that CPUs perform significantly worse than GPUs on highly parallelised workloads. The available processing power also limits the maximum input size, a too big input size will decrease performance (because of limited cache sizes). Because of the special vector instructions, the CPU does take advantage of vectorising when the input size is big enough.

As opposed to CPUs, the performance of GPUs doesn’t take advantage of vectorising. Vector instructions are missing in GPUs, but GPUs are designed in such a way that memory latency is hidden as much as possible. This is done by executing multiple warps (or wavefronts for AMD) on a streaming multiprocessor, when a warp has to wait on the memory another warp is executed on the streaming multiprocessor. Because of this design the approach for optimising for a GPU is entirely different than optimising for a CPU. It’s important to make sure the multiprocessors and CUDA cores of a GPU are constantly performing computations.

The number of threads does influence the performance significantly. Both for the CPU as for GPUs increasing the number of threads, increases performance.

We’ve also concluded that auto-tuning is effective, even for already optimised kernels as the Black-Scholes kernel we’ve used. Performance increases from 8 to 11% for GPUs and 44% for the CPU were visible.
Bibliography


Appendix

# Hardware specifications

Device 0: "GeForce GTX TITAN X"

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Driver Version / Runtime Version</td>
<td>8.0 / 8.0</td>
</tr>
<tr>
<td>CUDA Capability Major/Minor version number</td>
<td>5.2</td>
</tr>
<tr>
<td>Total amount of global memory</td>
<td>12207 MBytes</td>
</tr>
<tr>
<td>Total amount of global memory (bytes)</td>
<td>12799574016</td>
</tr>
<tr>
<td>Multiprocessors, CUDA Cores/MP</td>
<td>3072 CUDA Cores</td>
</tr>
<tr>
<td>GPU Max Clock rate</td>
<td>1076 MHz (1.08 GHz)</td>
</tr>
<tr>
<td>Memory Clock rate</td>
<td>3505 Mhz</td>
</tr>
<tr>
<td>Memory Bus Width</td>
<td>384-bit</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>3145728 bytes</td>
</tr>
<tr>
<td>Maximum Texture Dimension Size (x,y,z) 1D</td>
<td>65536, 2D (65536, 65536)</td>
</tr>
<tr>
<td>Maximum Layered 1D Texture Size, (num) layers</td>
<td>1D=(65536), 2D (16384), 2048</td>
</tr>
<tr>
<td>Maximum Layered 2D Texture Size, (num) layers</td>
<td>2D=(16384, 16384), 2048</td>
</tr>
<tr>
<td>Total amount of constant memory</td>
<td>65536 bytes</td>
</tr>
<tr>
<td>Total amount of shared memory per block</td>
<td>49152 bytes</td>
</tr>
<tr>
<td>Total number of registers available per block</td>
<td>65536</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of threads per multiprocessor</td>
<td>2048</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>1024</td>
</tr>
<tr>
<td>Max dimension size of a thread block (x,y,z)</td>
<td>(1024, 1024, 64)</td>
</tr>
<tr>
<td>Max dimension size of a grid size (x,y,z)</td>
<td>(2147483647, 65535, 65535)</td>
</tr>
<tr>
<td>Maximum memory pitch</td>
<td>2147483647 bytes</td>
</tr>
<tr>
<td>Texture alignment</td>
<td>512 bytes</td>
</tr>
<tr>
<td>Concurrent copy and kernel execution, engine(s)</td>
<td>Yes with 2 copy</td>
</tr>
<tr>
<td>Run time limit on kernels</td>
<td>No</td>
</tr>
<tr>
<td>Integrated GPU sharing Host Memory</td>
<td>No</td>
</tr>
<tr>
<td>Support host page-locked memory mapping</td>
<td>Yes</td>
</tr>
<tr>
<td>Alignment requirement for Surfaces</td>
<td>Yes</td>
</tr>
<tr>
<td>Device has ECC support</td>
<td>Disabled</td>
</tr>
<tr>
<td>Device supports Unified Addressing (UVA)</td>
<td>Yes</td>
</tr>
<tr>
<td>Device PCI Domain ID / Bus ID / location ID</td>
<td>0 / 3 / 0</td>
</tr>
</tbody>
</table>
Device 0: "TITAN X (Pascal)"
CUDA Driver Version / Runtime Version: 8.0 / 8.0
CUDA Capability Major/Minor version number: 6.1
Total amount of global memory: 12189 MBytes
(1278151616 bytes)
(28) Multiprocessors, (128) CUDA Cores/MP: 3584 CUDA Cores
GPU Max Clock rate: 1531 MHz (1.53 GHz)
Memory Clock rate: 5005 Mhz
Memory Bus Width: 384-bit
L2 Cache Size: 3145728 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(131072, 65536), 2D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 2 copy engines
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes
Device PCI Domain ID / Bus ID / location ID: 0 / 130 / 0

Device 0: "Tesla K20m"
CUDA Driver Version / Runtime Version: 8.0 / 8.0
CUDA Capability Major/Minor version number: 3.5
Total amount of global memory: 5061 MBytes
(5306777600 bytes)
(13) Multiprocessors, (192) CUDA Cores/MP: 2496 CUDA Cores
GPU Max Clock rate: 706 MHz (0.71 GHz)
Memory Clock rate: 2600 Mhz
Memory Bus Width: 320-bit
L2 Cache Size: 1310720 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(4096, 4096, 4096)
3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 2 copy engine(s)
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page–locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes
Device PCI Domain ID / Bus ID / location ID: 0 / 3 / 0

Device 0: "Tesla K40c"
CUDA Driver Version / Runtime Version: 8.0 / 8.0
CUDA Capability Major/Minor version number: 3.5
Total amount of global memory: 12205 MBytes
(12797607936 bytes)
(15) Multiprocessors, (192) CUDA Cores/MP: 2880 CUDA Cores
GPU Max Clock rate: 745 MHz (0.75 GHz)
Memory Clock rate: 3004 Mhz
Memory Bus Width: 384−bit
L2 Cache Size: 1572864 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 2 copy engine(s)
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page–locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes
Device PCI Domain ID / Bus ID / location ID: 0 / 3 / 0
Plots

Figure 7.1: The performance of the BlackScholes kernel using an input size of 80000 options with a varying number of threads and vector size.

Figure 7.2: The performance of the BlackScholes kernel using an input size of 160000 options with a varying number of threads and vector size.
Figure 7.3: The performance of the BlackScholes kernel using an input size of 320000 options with a varying number of threads and vector size.

Figure 7.4: The performance of the BlackScholes kernel using an input size of 640000 options with a varying number of threads and vector size.
Figure 7.5: The performance of the BlackScholes kernel using an input size of 1280000 options with a varying number of threads and vector size

Figure 7.6: The performance of the BlackScholes kernel on the Titan X (Pascal) using an input size of 80000 options with a varying number of threads and vector size
Figure 7.7: The performance of the BlackScholes kernel on the Titan X (Pascal) using an input size of 160000 options with a varying number of threads and vector size.

Figure 7.8: The performance of the BlackScholes kernel on the Titan X (Pascal) using an input size of 320000 options with a varying number of threads and vector size.
Figure 7.9: The performance of the BlackScholes kernel on the Titan X (Pascal) using an input size of 640000 options with a varying number of threads and vector size

Figure 7.10: The performance of the BlackScholes kernel on the Titan X (Pascal) using an input size of 1280000 options with a varying number of threads and vector size
Figure 7.11: The performance of the BlackScholes kernel on the K40 using an input size of 80000 options with a varying number of threads and vector size.

Figure 7.12: The performance of the BlackScholes kernel on the K40 using an input size of 160000 options with a varying number of threads and vector size.
Figure 7.13: The performance of the BlackScholes kernel on the K40 using an input size of 320000 options with a varying number of threads and vector size.

Figure 7.14: The performance of the BlackScholes kernel on the K40 using an input size of 640000 options with a varying number of threads and vector size.
Figure 7.15: The performance of the BlackScholes kernel on the K40 using an input size of 1280000 options with a varying number of threads and vector size.

Figure 7.16: The performance of the BlackScholes kernel on the E5-2630 using an input size of 80000 options with a varying number of threads and vector size.
Figure 7.17: The performance of the BlackScholes kernel on the E5-2630 using an input size of 160000 options with a varying number of threads and vector size.

Figure 7.18: The performance of the BlackScholes kernel on the E5-2630 using an input size of 320000 options with a varying number of threads and vector size.
Figure 7.19: The performance of the BlackScholes kernel on the E5-2630 using an input size of 640000 options with a varying number of threads and vector size

Figure 7.20: The performance of the BlackScholes kernel on the E5-2630 using an input size of 1280000 options with a varying number of threads and vector size