Norwegian University of
Science and Technology

Optimizing a High Energy Physics (HEP) Toolkit on Heterogeneous Architectures

Yngve Sneen Lindal

Master of Science in Computer Science

Submission date:	August 2011
Supervisor:	Anne Cathrine Elster, IDI
Co-supervisor:	Sverre Jarp, CERN

Norwegian University of Science and Technology
Department of Computer and Information Science

Optimizing a High Energy Physics (HEP) Toolkit on Heterogeneous Architectures

Yngve Sneen Lindal

NTNU

Page intentionally left blank

Abstract

A desired trend within high energy physics is to increase particle accelerator luminosities, leading to production of more collision data and higher probabilities of finding interesting physics results. A central data analysis technique used to determine whether results are interesting or not is the *maximum likelihood method*, and the corresponding evaluation of the *negative log-likelihood*, which can be computationally expensive. As the amount of data grows, it is important to take benefit from the parallelism in modern computers. This, in essence, means to exploit vector registers and all available cores on CPUs, as well as utilizing co-processors as GPUs.

This thesis describes the work done to optimize and parallelize a prototype of a central data analysis tool within the high energy physics community. The work consists of optimizations for multicore processors, GPUs, as well as a mechanism to balance the load between both CPUs and GPUs with the aim to fully exploit the power of modern commodity computers. We explore the OpenCL standard thoroughly and we give an overview of its limitations when used in a large real-world software package. We reach a single-core speedup of $\sim 7.8x$ compared to the original implementation of the toolkit for the physical model we use throughout this thesis. On top of that follows an increase of $\sim 3.6x$ with 4 threads on a commodity Intel processor, as well as almost perfect scalability on NUMA systems when thread affinity is applied. GPUs give varying speedups depending on the complexity of the physics model used. With our model, price-comparable GPUs give a speedup of $\sim 2.5x$ compared to a modern Intel CPU utilizing 8 SMT threads.

The balancing mechanism is based on real timings of each device and works optimally for large workloads when the API calls to the OpenCL implementation impose a small overhead and when computation timings are accurate.

Page intentionally left blank

Acknowledgements

I want to thank my supervisor, Dr. Anne C. Elster for taking the time to be my supervisor despite her sabbatical this year, for giving good advice, and for putting me in contact with, and recommending me to CERN in the first place. I also want to thank Else Lervik for recommending me to CERN.

I am very grateful to my colleagues at CERN openlab; my co-supervisor Sverre Jarp, Dr. Alfio Lazzaro, Julien Leduc and Andrzej Nowak for an inspiring working environment and constructive feedback on my work. I am especially grateful to Dr. Alfio Lazzaro which has been my closest mentor during this work. His help and feedback has been invaluable. I also want to thank my office colleague Julien Leduc once more for helping out with various software and hardware setups, in addition to providing useful comments on some of my work. Finally I want to thank Dr. Ian Karlin for reviewing my work and providing valuable corrections, and Filippo Spiga for providing access to an NVIDIA Tesla C2050 GPU during benchmarks.

Page intentionally left blank

Contents

Abstract **i**

Acknowledgements **ii**

List of tables **vii**

List of listings **viii**

List of figures **ix**

1 Introduction **1**

1.1 Motivation, goals and questions 3
1.2 Outline 5

2 Background **9**

2.1 The RooFit toolkit 9
2.2 The maximum likelihood method 12
2.3 Implementation 16
2.4 Optimization efforts 18
2.5 Principles of parallel computation 21
2.5.1 Flynn's taxonomy 21
2.5.2 Amdahl's law vs. Gustafsson's law 22
2.6 Numerical sensitivity 23
2.7 Eta'K model description 25

3 MLFit on multi-core processors **27**

3.1 The motivation for multi-core processors 27
3.2 Alternative threading technologies 29

3.2.1 Brief OpenCL introduction . . . 29
3.2.2 OpenCL for CPUs . . . 30
3.2.3 Preliminary OpenCL conclusion . . . 35
3.2.4 Intel Threading Building Blocks . . . 36
3.3 General optimizations . . . 38
3.3.1 A different evaluation approach . . . 38
3.3.2 Aiming for scalability . . . 41
3.3.3 Result propagation and loop fusion . . . 43
3.3.4 Constant expressions . . . 44
3.4 Results . . . 45
3.4.1 TBB and OpenMP, explicit and implicit . . . 47
3.4.2 Block splitting . . . 49
3.4.3 Scalability so far . . . 51
3.4.4 Result propagation, loop fusion and constant expressions . . . 54
3.4.5 NUMA results . . . 56
3.4.6 Preliminary conclusion . . . 57

4 MLFit on GPUs 59
4.1 Graphics processing units . . . 59
4.2 Implementation . . . 61
4.3 An initial experiment . . . 62
4.3.1 Measuring the double-precision basic operations . . . 64
4.3.2 Performance of computations involving transcendentals . . . 65
4.3.3 The effect of OpenCL vector types . . . 66
4.4 Optimization possibilities . . . 69
4.4.1 Single-precision . . . 70
4.4.2 Parallel reduction . . . 70
4.4.3 Texture cache . . . 71
4.4.4 Result propagation and loop fusion . . . 71
4.4.5 Constant expressions . . . 72
4.4.6 Occupancy . . . 72
4.5 Results . . . 73

5 Heterogeneous load balancing on commodity machines 79
5.1 Load balancing . . . 79

5.2 Strategy 80
5.2.1 Method 82
5.2.2 Implementation details 82
5.3 Benchmark results 85
5.3.1 Test case 1: Intel Core i7 965 + NVIDIA GTX470 85
5.3.2 Test case 2: Intel Core i7 965 + AMD Radeon HD5870 88
5.3.3 Test case 3: NVIDIA GTX470 + AMD Radeon HD5870 91
5.4 Considerations on optimal execution configurations 93

6 Conclusions and future work 95
6.1 Conclusions 95
6.2 Future work 98

A OpenCL test kernels with varying arithmetic intensity 105

B OpenCL test kernels involving transcendentals 109

C The eta'K model implementation 111

D A tree illustration of the eta'K model 117

E OpenCL kernels for the PDFs used in the eta'K model 119

F NVIDIA Tesla benchmarks 123

G CERN openlab report: *First encounter with OpenCL for multicore CPUs* 127

List of Tables

3.1 5 timings for 100 eta'K model evaluations with 4 threads and 1 000 000 events. 47

4.1 A specification comparison between the NVIDIA GeForce GTX470 and the AMD Radeon HD5870. 63
4.2 DP basic operations counts for the kernels listed in Appendix A. 64
4.3 Timings, mean, standard deviation and standard error for 100 *NLL* evaluations with 1 000 000 events, both for the GTX470 and the HD5870. 77

5.1 5 timings, mean, standard deviation and standard error for the balanced run with the Core i7 965 and the GTX470. 1 000 000 events. 86
5.2 5 timings, mean, standard deviation and standard error for the balanced run with the Core i7 965 (3 threads) and the HD5870. 1 000 000 events. 89
5.3 5 timings, mean, standard deviation and standard error for the balanced run with the GTX470 and the HD5870. 1 000 000 events. 91
5.4 A comparison between equation 5.4.2 and some well-known polynomials. . . 93

F.1 The specifications of the NVIDIA Tesla C2050 professional GPU. 123

List of listings

2.1 A RooFit program to plot toy data on a pre-defined statistical model. 9
2.2 A simplified version of the base PDF class of RooFit. 16
2.3 A simple implementation of the *NLL* function, relying on a PDF composite model. 17
3.1 OpenMP version of a Gaussian evaluation function. 30
3.2 The OpenCL equivalent of the function in listing 3.1. 30
3.3 A wrapper function for calling the OpenCL Gaussian evaluation kernel in listing 3.2. 32
3.4 The vectorized version of the kernel in listing 3.2. 34
3.5 Same as listing 3.4, but with more work per thread. 35
3.6 C++/pseudocode describing the use of the TBB *parallel_for* method. 37
3.7 Evaluation of the *BifurGaussian* function, calculating constant values for each evaluation. 45
5.1 A conceptual implementation of the balancing method. 83
5.2 Threading implementation utilizing both CPU(s) and GPU(s). 84
A.1 OpenCL kernels with varying arithmetic intensity 105
B.1 OpenCL test kernels involving transcendentals 109
C.1 The eta'K model implementation 111
E.1 OpenCL kernels for the PDFs used in the eta'K model 119

List of Figures

1.1 The Atlas detector. ATLAS Experiment © 2011 CERN. 2
1.2 One signal and one background event probability distribution fitted to toy data, and the combined signal+background PDF fitted to real data. 4

2.1 The resulting plot from the program in Listing 2.1. 11
2.2 The search for the minimum of a 2D function. 15
2.3 A simple illustration of the PDF tree resulting from the listing in Figure 2.1. 17
2.4 An Intel Vtune Amplifier 2011 hotspot profile of the optimized OpenMP version of the MLFit application running with 100 000 events and 4 threads on an Intel Core i7 965 machine. Note that this is an excerpt of the contributing functions, showing the most time consuming ones. 20
2.5 An illustration of Amdahl's law for a parallel computation with serial fractions of respectively 5%, 10%, 25% and 50%. 23

3.1 An OpenCL thread grid consisting of workgroups (or *blocks* in NVIDIA terminology). 29
3.2 Performance comparison of a single-threaded evaluation of the eta'K model for OpenCL versus OpenMP with the Intel C compiler both with and without vectorization (both SSE and AVX). 1 000 000 events. 36
3.3 An explicitly parallel evaluation of a PDF tree. 39
3.4 An implicitly parallel evaluation of a PDF tree. 40
3.5 VTune hotspot profiles for the OpenMP implicitly parallel version on 4 threads. N is 100 000 and 1 000 000 respectively. 41
3.6 The topology of an Intel Core i7 965 CPU. 46
3.7 OpenMP version versus TBB version on the Intel Core i7 965. Both explicitly parallel and implicitly parallel. 48

3.8 Implicitly parallel versions of OpenMP and TBB on the Intel Core i7 965, with and without block splitting. The blocked explicitly parallel version is also included to compare with the blocked implicitly parallel one. 50
3.9 The average speedup of going from an explicitly parallel to an implicitly parallel evaluation, with errors. The average is taken over all threads, and is shown for different values of N. 51
3.10 Scalability of the OpenMP blocked implicitly parallel version. 53
3.11 Scalability of the most important OpenMP versions. 55
3.12 Speedup relative to the OpenMP explicit version. 55
3.13 MLFit scalability on a dual-socket Intel Core i7 machine. 57
3.14 A VTune hotspot profile of the final version, running 1 000 000 events on 4 threads on the Intel Core i7 965. 58

4.1 An explicitly parallel evaluation with OpenCL in MLFit. 62
4.2 The benchmark results for both NVIDIA GTX470 and AMD Radeon HD5870 running the kernels in Appendix A. 65
4.3 Benchmark results from the kernels in Appendix B running on the NVIDIA GTX470 and the AMD Radeon HD5870 respectively. 67
4.4 Kernel 6 from appendix A run with and without using double-precision vector types for both the GTX470 and the HD5870. 69
4.5 Kernel 6 from appendix A run with and without using single-precision vector types for both the GTX470 and the HD5870. 70
4.6 MLFit benchmark, comparison between CPU and GPU. 74
4.7 MLFit benchmark, comparison between CPU and both GPUs. 76
4.8 Gaussian function evaluation, CPU and both GPUs. 76

5.1 The balancing convergence of the Core i7 965 and the GTX470 together, running with 1 000 000 events. 86
5.2 Speedup results from balancing between the Core i7 965 and the GTX470. . 87
5.3 The balancing convergence of the Core i7 965 and the HD5870 together, running with 1 000 000 events. 89
5.4 Speedup results from balancing between the Core i7 965 and the Radeon HD5870. 90
5.5 Speedup results from balancing between the GTX470 and the Radeon HD5870. 92

D.1 Tree illustration of the eta'K model. Double-circled red nodes are composite PDFs, single-circled are ordinary PDFs and blue nodes are variables. 118

F.1 Equivalent to figure 4.2, but with the Tesla C2050 in addition. 124
F.2 Tesla results for the kernels involving transcendentals in Appendix B. . . . 125
F.3 MLFit benchmark including results for the Tesla C2050. Eta'K model evaluation. 126

Chapter 1

Introduction

Particle physics, or High Energy Physics (HEP), is the field of studying the basic building blocks of the universe, the elementary particles. These particles includes 6 quarks, 6 leptons, intermediate bosons, and hadrons, as well as all these particles' antiparticles [1]. In addition, theorists believe there are yet undiscovered particles like for instance the Higgs boson(s). For each of these undiscovered particles, there exist important questions; does it exist? what are its physical properties? what happens when it collides with another particle? etc. However, many particles like the elementary particles cannot be seen in the natural physical conditions on our planet, but can only be discovered and studied under very special circumstances. There is ongoing research to find new particles, as well as study other effects, and to accomplish that, subatomic particles like protons and ions are accelerated by extreme electromagnetic fields and collided against each other in special machines called particle accelerators. Then, to catch instances of some phenomenon X, often enormous devices called detectors tries to observe the effect of particle interactions/decays (also called *events*), interpret physical results, and store them as binary data.

CERN, the *European Organization for Nuclear Research* is an international organization with the purpose to operate the world's largest particle physics laboratory [2]. It is located just outside Geneva at the border between France and Switzerland. It is the birthplace of the World Wide Web and also the home of the Large Hadron Collider (LHC), which is the world's largest and most high-energetic particle accelerator [3]. The LHC can accelerate beams of particles with so high energy levels that hopefully new physics extending the Standard Model of particle physics [4] (the model describing the dynamics of sub-atomic particles) can be found, as well as answers to remaining unsolved questions within physics. The Atlas detector at the LHC, reprinted from [5], is illustrated in Figure 1.1. The amount of data produced

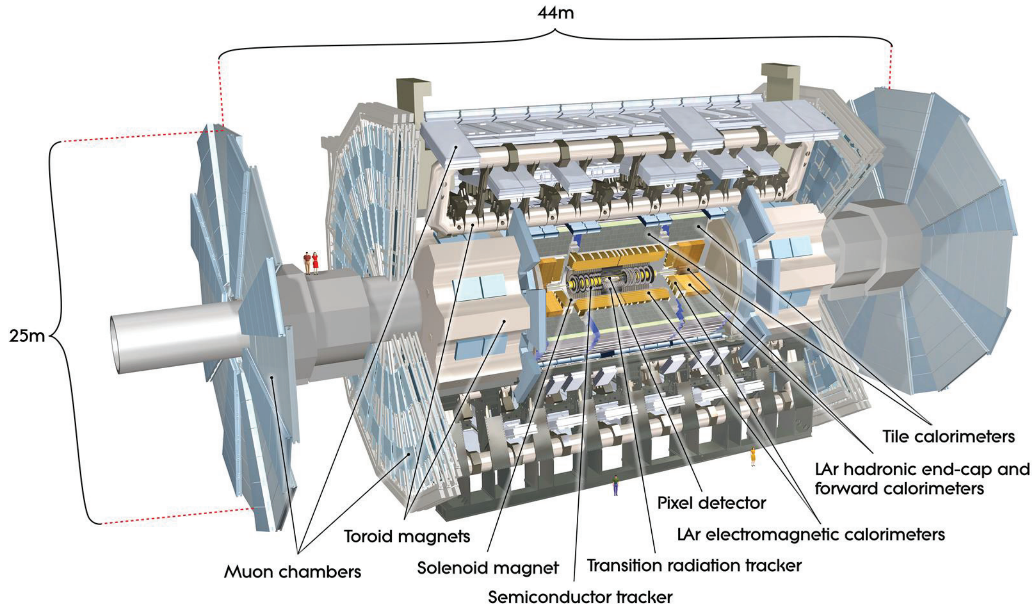


Figure 1.1: The Atlas detector. ATLAS Experiment © 2011 CERN.

by collision detectors is huge, and therefore a lot of computing power in terms of number-crunching capability as well as storage capacity is needed.

A collision, as we mentioned, can lead to observations/phenomena called events. An event is a collection of physical parameters from the collision. The detectors select the events that look promising (also called *candidate events*) with respect to what is expected to be found and record them. An experiment often starts with a physical model of the expected outcome, which in essence is a collection of probability density functions (hereby abbreviated PDFs) combined in some way, each of them representing some probability distribution for e.g. which energies should be observed if an instance of phenomenon X has been observed, which velocities, which masses etc. When these candidate events are recorded, they can be analyzed further by statistical analysis bound to the relevant model. Events that to a high statistical certainty are believed to be instances of X are called *signals,* while all other events are called *backgrounds.* There will always be background events in an experiment since the probability is very high for that other interactions/decays that are not directly relevant for X happen. Figure 1.2a shows a signal PDF and figure 1.2b shows a background PDF. The dots in these two plots are just toy data generated by Monte-Carlo methods with respect to the PDF. However, figure 1.2c shows a PDF combined of these two fitted to *real* data. It is easy to see that the observations fit the model good, and that the probability of finding

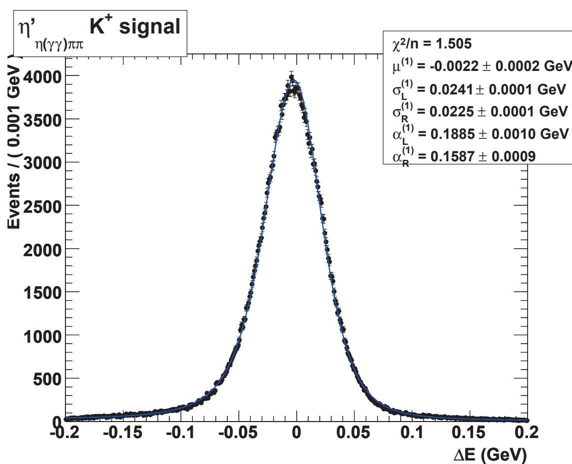
what was expected is high.

The work of this thesis has been performed during the spring of 2011 at CERN openlab, which is a collaboration between CERN and industrial partners to develop new knowledge in Information and Communication Technologies through the evaluation of advanced solutions and joint research to be used by the worldwide community of scientists working at the Large Hadron Collider [6].

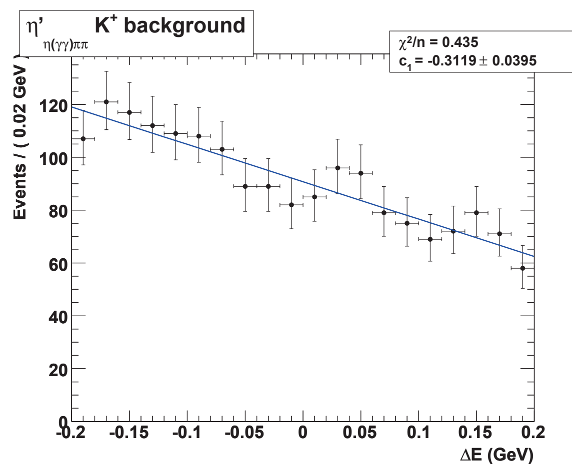
1.1 Motivation, goals and questions

The physical model we have used for testing and benchmarking throughout this thesis is a model stemming from the publication *B meson decays to charmless meson pairs containing η or η´ mesons* [7], which is a real measurement performed by the *BABAR* collaboration at the SLAC National Accelerator Laboratory (SNAL) [8]. The model will hereby be called *eta'K* because of the measurement it was used for, that is, *B meson decay to η'K final state.* To give the main motivation of this work, we give an example by introducing the concepts of cross section and luminosity. A *barn* is a *unit of area used to measure the reaction cross section of atomic nuclei and subatomic particles in the study of their interactions with other nuclei or particles* [9], and equals $10^{-24}\,cm^2$. Luminosity is a measure of the total data collected by an accelerator, and can be expressed by *inverse barn*, b^{-1} and inverse time, s^{-1}. If one integrates luminosity with respect to time, one gets the *integrated luminosity*, expressed by inverse barn. The η'*K* measurement in [7] aimed to gather 130 000 candidate events, and was performed at an integrated luminosity of 426 fb^{-1} (426 inverse femtobarn, or $426 * 10^{15}$ inverse barn), with the PEP-II accelerator at SNAL. According to [10], *SuperB* is a new accelerator that aims to reach an integrated luminosity of 50 ab^{-1} (50 inverse attobarn, $50 * 10^{18}$ inverse barn or 50 000 fb^{-1}). This will in principle mean a potential of gathering $\sim$ 100 times as many events, and with the experiment we used as an example, this could mean $\sim$ 13 000 000 events as opposed to 130 000 events. This is a general trend, also at the LHC [11], so more and more data will be produced and thereby needs to be analyzed.

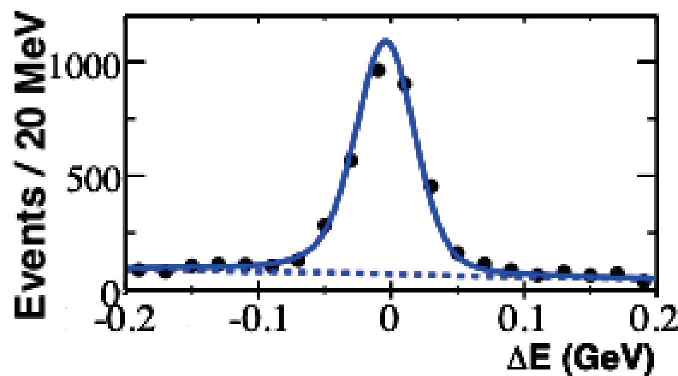
The methods used for analyzing these events are often very computationally expensive, which we will see more clearly in chapter 2. It becomes therefore more and more important as accelerators gets more and more powerful and reaches higher luminosities, to exploit the available computing hardware that these methods are run on to a high degree. We add



(a) A signal PDF.



(b) A background PDF.



(c) The combined signal+background PDF.

Figure 1.2: One signal and one background event probability distribution fitted to toy data, and the combined signal+background PDF fitted to real data.

that these final analyses are often carried out on the personal computers of physicists, i.e. commodity machines.

With these things in mind, the primary aim of our work is first of all to do elementary optimizations to a toolkit (RooFit) used for analyzing the events from physics detectors, so that these analyses could be performed as fast as possible on commodity machines, with the constraints of RooFit in mind. We also look into the use of accelerators as graphics processing units (GPUs) and their suitability for this task, as well as trying to exploit all computational resources in one machine at the same time (CPUs and GPUs), in a hopefully simple and elegant way. The work is a continuation of an already started process of optimization and parallelization of this program package. Performance is not the only thing essential, though. Since the program is large and written in a such a high-level language as C++, achieving a high degree of programmability is also paramount. There exists huge amounts of code using the interfaces of this package, so rule number one is to keep the interface consistent to the highest degree possible. Based on these aims, our primary research questions are;

- Which optimizations can be done to RooFit's data analysis methods without changing the program structure/interface substantially?
- Is it feasible to use the OpenCL standard, by the current implementations of it, for programming both CPUs and GPUs in a large real-world C++ software package that makes use of high-level programming language constructs?
- If not, what is a simple and effective way of exploiting the available hardware?

We believe much of the strength of these questions lies in the fact that this is a large, real-world software package, and not just an academic software example. We thus feel that our answers to these questions can be interesting to other programmers within the HEP community that deal with parallelism, but also to some degree to every programmer that deals with real-world applications on heterogeneous commodity systems, or larger systems for that matter.

1.2 Outline

This thesis is structured as follows.

Chapter 2 (Background) Gives an introduction to the data analysis toolkit around which this thesis is centered, and the mathematical foundation of the main algorithm we focus on.

In addition, previous work, special areas needed to be taken care of, as well as some background within parallel computing are covered.

Chapter 3 (MLFit on multi-core processors) Describes in details the work done to optimize the prototype MLFit for fast execution and good scalability on commodity multi-core processors.

Chapter 4 (MLFit on GPUs) Focuses on the development of an OpenCL version of MLFit to be run on GPUs, as well as a general performance comparison between two GPUs from different vendors.

Chapter 5 (Heterogeneous load balancing on commodity machines) Gives an introduction to load balancing, which constraints and challenges MLFit imposes on this field and a simple method for balancing different computational devices based on real timings. Results are described and explained and selection of optimal runtime configurations are discussed.

Chapter 6 (Conclusions) Concludes the work by presenting and reflecting upon the findings.

Appendix A (OpenCL test kernels with varying arithmetic intensity) Lists a few OpenCL computational kernels used to explore the theoretical peak performance for the two main GPUs we use in this thesis.

Appendix B (OpenCL test kernels involving transcendentals) Lists a few OpenCL computational kernels used to determine performance characteristics when computing trascendental and power functions.

Appendix C (The eta'K model implementation) Lists the C++ implementation of the main physical model we use for performance analysis in this thesis.

Appendix D (A tree illustration of the eta'K model) Shows an illustration of the eta'K model.

Appendix E (OpenCL kernels for the PDFs used in the eta'K model) Lists the OpenCL kernels used for the PDFs in the eta'K model.

Appendix F (NVIDIA Tesla benchmarks) Shows benchmarks using an NVIDIA Tesla C2050 professional GPU compared to the commodity GPUs we use in this thesis.

Appendix G (First encounter with OpenCL for multicore CPUs) An experience/feedback report primarily intended for Intels OpenCL group written during our work at CERN openlab.

Chapter 2

Background

This chapter gives an introduction to the RooFit toolkit [12], as well as the mathematical foundation for the algorithm used to fit statistical parameters to collected data. Some principles of parallel computation are mentioned, and an overview of the optimizations that has already been done to this package are given. We also mention some important restrictions regarding the fitting procedure.

2.1 The RooFit toolkit

RooFit is a toolkit for modeling event distributions in particle physics experiments and is integrated with the ROOT system [13]. ROOT is a set of object oriented frameworks used to handle large amounts of data in an efficient way. It includes tools for doing histogramming, curve fitting, function evaluation, minimization, graphics and visualization. RooFit uses in particular ROOT's graphics packages to visualize data and model plots. ROOT and RooFit are written in C++, and make extensive use of the features of the language (polymorphism and the use of virtual functions in particular). RooFit is able to fit data using a variety of techniques, and we will in this thesis focus on one of them; *the maximum likelihood method*.

Recall from Chapter 1 the concepts of signal and background events. RooFit provides methods to approximate which events are signals, and which are backgrounds. Physicists start with a prediction of a model of the distribution of different events. This will differ from measurement to measurement, and will involve PDFs for each quantity of interest. One of RooFit's powerful features is the object-oriented way to build composed PDFs from a set of base PDFs. The resulting PDF could be viewed as a tree of PDFs, and virtual functions are used to evaluate each PDF, since every PDF is derived from a base class which contains

Listing 2.1: A RooFit program to plot toy data on a pre-defined statistical model.

```
// --- Observable ---
RooRealVar mes("mes","m_{ES} (GeV)",5.20,5.30);

// --- Build Gaussian signal PDF ---
RooRealVar sigmean("sigmean","B^{#pm} mass",5.28,5.20,5.30);
RooRealVar sigwidth("sigwidth","B^{#pm} width",0.0027,0.001,1.);
RooGaussian signal("signal","signal PDF",mes,sigmean,sigwidth);

// --- Build Argus background PDF ---
RooRealVar argpar("argpar","argus shape parameter",-20.0,-100.,-1.);
RooArgusBG background("background","Argus PDF",mes,RooFit::RooConst(5.291),argpar);

// --- Construct signal+background PDF ---
RooRealVar nsig("nsig","#signal events",200,0.,10000);
RooRealVar nbkg("nbkg","#background events",800,0.,10000);
RooAddPdf model("model","g+a",RooArgList(signal,background),RooArgList(nsig,nbkg));

// --- Generate a toyMC sample from composite PDF ---
RooDataSet *data = model.generate(mes,2000);

// --- Declare the NLL ---
RooNLLVar nll("nll","NLL",model,*data,RooFit::Extended());

// --- Send the NLL to the minimizer (Minuit) ---
RooMinimizer minimizer(nll);
minimizer.optimizeConst(true);
minimizer.setMinimizerType("Minuit");

// --- Run the minimization ---
// --- Perform extended ML fit of composite PDF to toy data ---
minimizer.migrad();
minimizer.hesse();

// --- Plot toy data and composite PDF overlaid ---
RooPlot* mesframe = mes.frame() ;
data->plotOn(mesframe) ;
model.plotOn(mesframe) ;
model.plotOn(mesframe, RooFit::Components(background), RooFit::LineStyle(kDashed));
mesframe->Draw();
```

general methods (e.g. a method for evaluating the function with a given set of parameters).

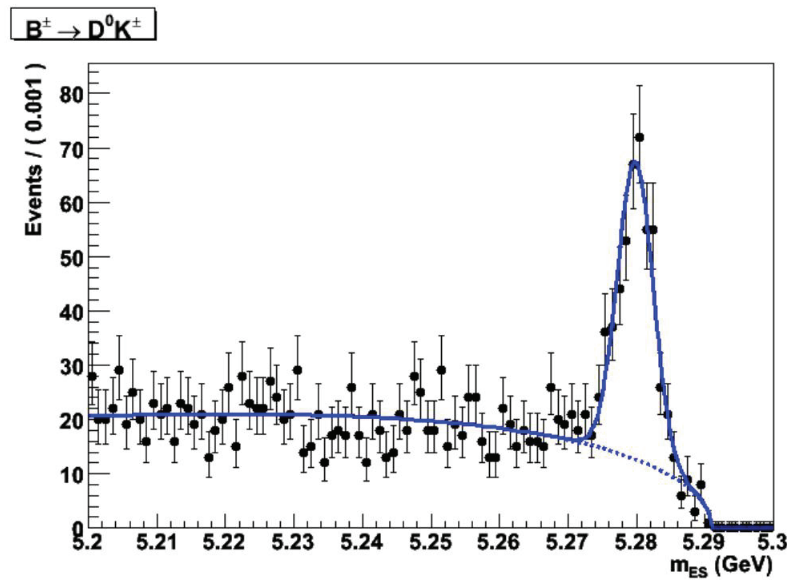


Figure 2.1: The resulting plot from the program in Listing 2.1.

Listing 2.1 shows a code example from the RooFit user manual that creates two PDFs, one Gaussian signal PDF and one Argus[1] background PDF. The total PDF is the sum of these two. Figure 2.1 shows the two PDFs representing signal and background event distributions respectively. The plotted points in this example are toy Monte-Carlo samples generated wrt. this model. The program then performs an extended maximum likelihood fitting of the function parameters to these toy data. As mentioned in Chapter 1, if one searches for a particular (maybe yet to be discovered) particle, the fitting procedure is essential to decide whether the data show signs of that particle or not, i.e. fits the model. In a real example, the samples above would of course be real samples taken from particle collision detectors. To translate to this example; if a significant portion of the events from an experiment each contribute a variable value that could be plotted approximately on top of the gaussian function, it would be a strong signal of what is expected, since the Gaussian in our case represents the signal we are interested in. Of course, sofisticated statistical methods are used to decide the threshold limits for when an event distribution could be deemed a discovery or not [15].

[1]An Argus PDF is a special PDF described in [14], and is defined as $x\sqrt{1-x^2}\exp\left[-\xi(1-x^2)\right]$, with $x \equiv 2m_{ES}/\sqrt{s}$ and ξ a parameter that is determined by the fit.

2.2 The maximum likelihood method

Recall the definition of an event as being a set of variables, measured at a given point in time by a particle collision detector. A data sample consists of N different, independent events. Following [16], an event could be represented as a multidimensional random vector of variables $\hat{x} = (x_1, ..., x_n)$ (masses, positions, energies etc.) that could be described by a PDF $\mathcal{P}(\hat{x}|\hat{\theta})$, where $\hat{\theta}$ is a set of p real parameters (read: *the propability for this set of variables, $\hat{x}$, given the parameter estimation $\hat{\theta}$*). We further assume that $\hat{x}$ is well known, so that $\mathcal{P}(\hat{x}|\hat{\theta})$ after normalizing[2] it represents a hypothesized PDF for $\hat{x}$. In an experiment, one conducts a series of N independent measurements leading to a data sample consisting of $\hat{\mathbf{x}} = \hat{x}_1, ..., \hat{x}_n$ values. The joint PDF of $\hat{\mathbf{x}}$ is by independence,

$$f(\hat{x}|\hat{\theta}) = \prod_{i=1}^{N} \mathcal{P}(\hat{x}_i|\hat{\theta}). \tag{2.2.1}$$

With uncorrelated variables, equation 2.2.1 can be written as a product of individual PDFs dependent on each variable,

$$\mathcal{P}(\hat{x}_i|\hat{\theta}) = \prod_{v=1}^{n} \mathcal{P}^v(x_i^v|\hat{\theta}). \tag{2.2.2}$$

If we replace $\hat{\mathbf{x}}$ with a collection of real observed data, f will no longer be a PDF. Since $\hat{\mathbf{x}}$ is now known, f is only dependent of $\hat{\theta}$, and is usually denoted $\mathcal{L}$, being the *likelihood function*. Thus, $\mathcal{L}(\hat{\theta}) = f(\hat{x}|\hat{\theta})$. This function can be used to estimate the set of parameters that *fits the total data sample the best way*, or in other words, finding the estimate $\hat{\theta}$ for which $\mathcal{L}(\hat{\theta})$ has its maximum. This procedure is called the *maximum likelihood (ML) method,* and is a popular statistical technique for this scenario [17]. It is possible to use the ML method not only to estimate $\hat{\theta}$, but also the number of events belonging to the different species in a data sample, i.e. signals or backgrounds. Given s different species and defining with n_j the number of events belonging to species j and with $\mathcal{P}_j(\hat{x}|\hat{\theta}_j)$ the PDF for the species j, the *extended likelihood function* is defined as:

$$\mathcal{L} = \frac{e^{-\sum_{j=1}^{s} n_j}}{N!} \prod_{i=1}^{N} \sum_{j=1}^{s} n_j \mathcal{P}_j(\hat{x}_i|\hat{\theta}_j), \tag{2.2.3}$$

[2]The evaluation of a PDF is completed by doing a normalization of the function value, to make the integral equal 1 (i.e. making the function a true probability density function).

which consists of an extended term to take in account that the number of observations N in the sample is itself a Poisson random variable with a mean value $\sum_{j=1}^{s} n_j$.

The search for the maximum of $\mathcal{L}$ can be done numerically. An often used method is to minimize the equivalent function $-ln(\mathcal{L})$, the *negative log-likelihood (NLL)*. The functions $f(x) = x$ and $f(x) = ln(x)$ are monotonic, so the search for the minimum would be an equivalent procedure for this new function. Ignoring the extended term and using the fact that $ln(a_1 * a_2 * ... * a_n) = ln(a_1) + ln(a_2) + ... + ln(a_n)$, the NLL to be minimized has the form

$$NLL = \sum_{j=1}^{s} n_j - \sum_{i=1}^{N} \left(ln \sum_{j=1}^{s} n_j \mathcal{P}_j(\hat{x_i}|\hat{\theta}_j) \right), \tag{2.2.4}$$

which clearly is a sum of logarithms. According to [18], the most common method used in the HEP community to minimize the NLL is based on the MIGRAD algorithm inside the MINUIT package (the C++ implementation is called MINUIT2). MIGRAD minimizes a function using a *variable metric method* [19], called the *Davidon-Fletcher-Powell* method [20]. This method involves the calculation of the derivatives of the NLL for each free parameter. According to [21], MIGRAD is also able to use approximation techniques like finite differences to approximate the first derivatives of the function if there exists no analytical differentiation, that is,

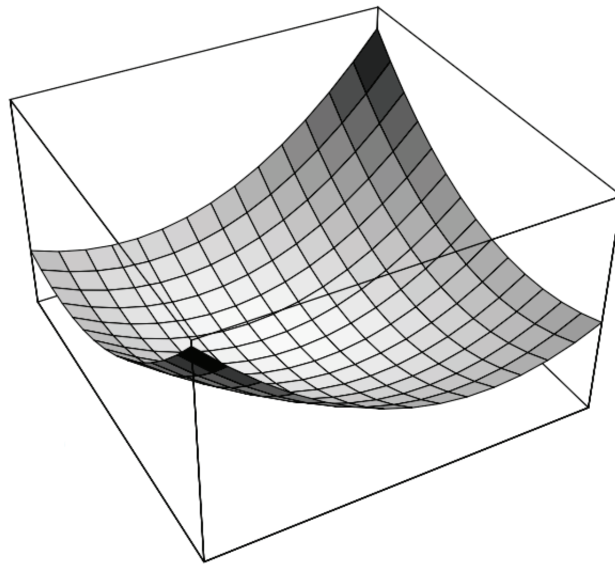
$$\frac{\partial NLL}{\partial \hat{\theta}} \Big|_{\hat{\theta}_0} \approx \frac{NLL(\hat{\theta}_0 + \hat{d}) - NLL(\hat{\theta}_0 - \hat{d})}{2\hat{d}}, \tag{2.2.5}$$

which means two calls to the NLL function for each variable. A numerical evaluation of the second derivatives is also required at the end of the minimization. Variable metric methods in general use Newtons method to find stationary points of functions where the gradient is 0, and thus in our case where

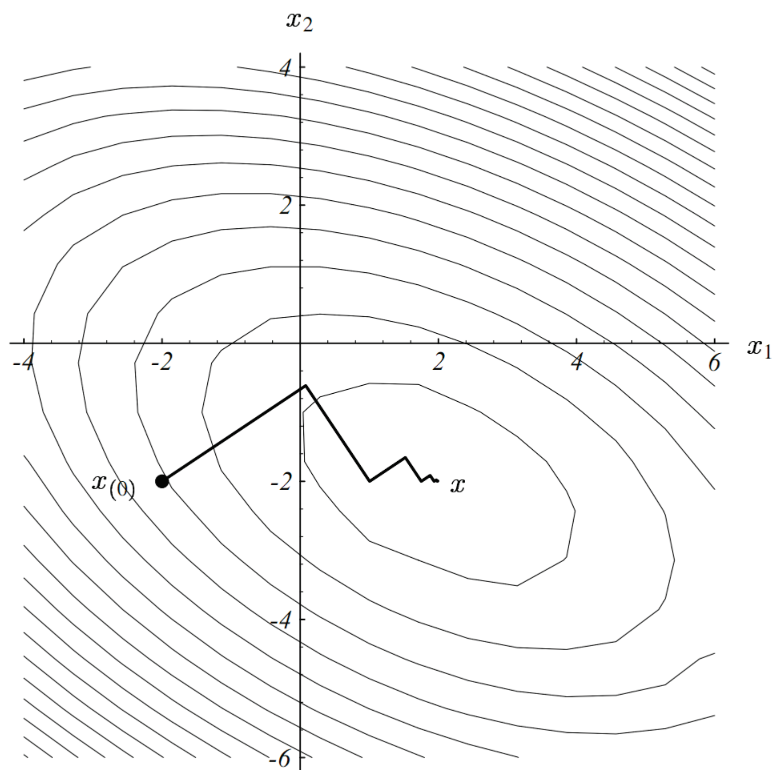
$$\frac{\partial NLL}{\partial \hat{\theta}} = 0, \tag{2.2.6}$$

which is called the *likelihood equation* [15], and would be the ultimate goal of the minimization routine. The nice feature of equation 2.2.3 (and 2.2.4) is that n_j is a tuneable parameter of the function, which can be used to determine the number of events in the different species, which again could be used to decide if the data sample to some degree of statistical significance could represent an interesting result/discovery. Since our work will not involve modifying MINUIT2 in any way, we will not delve deep into the theory behind these minimization algorithms. However, to give a feeling of what happens during minimization, a 2D function and the search for its minimum is illustrated graphically in Figure 2.2. These images are reprinted from [22]

(with permission from the author), which is a very instructive and pedagogical paper about function minimization. The method used in this case is the method of steepest descent, that uses the quadratic form of a matrix representing a system of equations to find the solution of it.



(a) A 2D function.



(b) The search for its minimum.

Figure 2.2: The search for the minimum of a 2D function.

2.3 Implementation

To perform the minimization procedure, MINUIT is dependent of the *NLL* function we described in Section 2.2. This function is implemented in a RooFit class called *RooNLLVar*, and uses a model object defined by the programmer. The model will of course vary for each analysis, and is represented by an instance of the class RooAbsPdf, which is the root node of the PDF model.

Listing 2.2: A simplified version of the base PDF class of RooFit.

```
class RooAbsPdf
{
protected:
  virtual Double_t evaluate() const = 0;
};
```

Listing 2.2 shows a very simplified version of the base PDF class of RooFit. It is up to derived classes to implement the pure virtual function evaluate, depending on what the characteristics of the given PDF are. The three PDF classes in listing 2.1; RooGaussian, RooArgusBG and RooAddPdf all inherit from this base class. Note that the class is a lot more comprehensive than this one; this illustration just tries to clarify the call graph that happens inside RooFit when a call to the *NLL* function occurs. When the program in listing 2.1 is run, a tree of PDFs will be created (see Figure 2.3), and the following will happen:

1. When generating the RooDataSet, the function “generate” will use the function RooAddPdf::evaluate().
2. RooAddPdf::evaluate() will call RooGaussian::evaluate() and RooArgusBG::evaluate() respectively, obtaining one result from each.
3. RooAddPdf::evaluate() sums these results, and returns the sum.

Of course, the tree can be of arbitrary size since models often can be very complex.

Recall the structure $\hat{\mathbf{x}}$ from Section 2.2. This structure is an array of arrays, where the inner arrays represents the different variables measured at one single event, making the row i represent event i. The procedure is illustrated in listing 2.3. The function (based on the *NLL* definition) takes a set of parameter estimates, $\hat{\theta}$. It then loops over all events, and for each event, calls the root PDF object with these parameters. This involves virtual function calls for each *NLL* call, for each event, for each PDF (and a model can consist of *many* PDFs). And as we already know, the number of events can become very high depending on

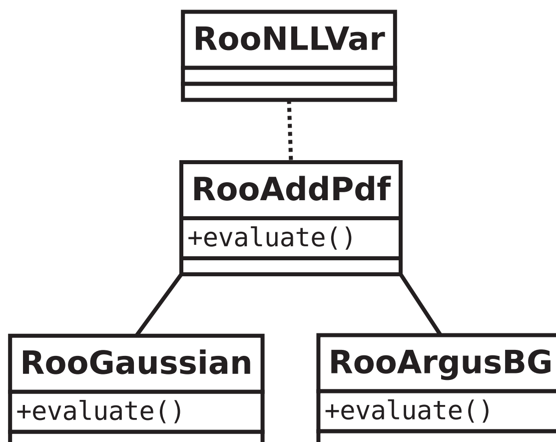


Figure 2.3: A simple illustration of the PDF tree resulting from the listing in Figure 2.1.

the experiment and the detectors, and it is also a number that is expected to increase, as we described in Chapter 1.

Listing 2.3: A simple implementation of the *NLL* function, relying on a PDF composite model.

```
Double_t NLL::GetVal(<parameters>, const UInt_t N)
{
  //Assume that we have a PDF tree in a AbsPdf object called "model"
  Double_t sum = 0.0;
  for(UInt_t i = 0; i < N; i++)
  {
    sum -= model.GetLogVal(<parameters>, i);
  }
  return sum;
}
```

The *NLL* function itself is evaluated numerous times (could be thousands) for different combinations of the parameters, hopefully converging to the minimum. This is obviously an implementation not having performance in mind. There are two obvious points that ruins the performance of this computation.

- A compiler would not be able to either vectorize the code effectively or perform inter-procedural analysis since code paths are not clear at compile-time because of the dynamic dispatch. And if the functions were not virtual, they would have to be inlined for the compiler to be able to vectorize efficiently.

- By calling the PDF tree and its virtual functions *for each* x, virtual function overhead

is built up.

In addition, assuming we have for instance three parameters, RooFit stores all parameter data in a matrix on the form $\theta_{1_i}\theta_{2_i}\theta_{3_i}, \theta_{1_{i+1}}\theta_{2_{i+1}}\theta_{3_{i+1}} \dots \theta_{1_n}\theta_{2_n}\theta_{3_n}$ suggesting bad locality when using one and only one of these parameters at a time. In general, this is a great example of a trade-off situation between programmability and performance. RooFit gives the programmer the expressive power to design models composed of virtually any PDF combination. These PDFs could also be customly defined by a newly introduced class inheriting from RooAbsPdf and implementing the function *evaluate* (and some others not included here). The price for this is a program that will not take advantage of the potential of modern CPUs or accelerators like GPUs, since the computation is not done on vectors, but on single values with e.g. conditional jumps between each value (much like an *array of structures* (AOS) instead of a *structure of arrays* (SOA)).

2.4 Optimization efforts

RooFit is an extensive software package. The *NLL* evaluation that we focus on is just a part of it, and we have therefore based our work on a prototype called MLFit. This is a small application made to be able to perform *NLL* fits, and is, like RooFit, compatible with MINUIT. It supports a subset of the PDF functions that RooFit offers, and this subset is sufficient for our work. This way we can perform realistic benchmarks on a much simpler software package.

The efforts done to optimize MLFit so far are described in [16] and [23]. The most important change to the implementation was to evaluate each PDF for all x in one single call. Instead of an evaluation function that returns *one* floating-point number, it now returns an *array* of floating point numbers. After all, the evaluation is the same, since the only difference is the argument x_i. This step is repeated, so in the end one would have a partial result array for each PDF. These are then combined in the PDFs responsible for e.g. adding or taking the product of two child PDFs (see Appendix C, listing the implementation of the eta'K model), which in the end would propagate up to the root as the final result. Then the logarithm of these values is calculated, and a negative reduction is applied on top of that to produce the single *NLL* value (see equation 2.2.4). This introduces two important benefits; it

- reduces the number of virtual function calls from $k * N$ to k where k is the number of PDFs, and N is the number of events. That is, the number of virtual function calls

is now independent of the number of events, and when N could be 1 000 000 or even maybe 13 000 000 as we predicted in Chapter 1, and this happens for each call to the NLL function, this is significant.

- structures the code in a manner that makes use of arrays, which again makes it easier for compilers to vectorize. The problem of not being able to do interprocedural analysis is now reduced significantly, since most of the time spent in calculations actually is done inside the evaluation loops. Because of the use of arrays, the code is now also easily parallelizable using multithreaded libraries like OpenMP for instance.

There has also been done other modifications to the code that has revealed speedups (like multiplying with reciprocals instead of division), but these are details compared to the structural changes. The results of all these optimizations are reported by Jarp et al. in [23] to be $\sim 4.5x$ on a single core compared to the original RooFit. It is also important to point out that there are not just benefits associated with these optimizations. A downside is that we now have to store vectors of results for each PDF, which obviously takes up more memory compared calculating the result "on the fly". This can be a limitation when the number of PDFs and N grow.

By introducing the new strategy for evaluation of the functions, OpenMP is a natural choice to spread work across cores. This is implemented by adding a virtual function *evaluateOpenMP* as a virtual function in the class RooAbsPdf. It is a quite elegant solution in terms of programmability, since the class encapsulates the details of the OpenMP specific code, and since one keeps the benefits of polymorphism through virtual functions. The threads are spread across the current array in a statically partitioned manner. This means that each thread has an approximate equal size of the array, and it is implemented in a way so that one thread can never have more than one element more than another thread, to ensure an equal load balancing. Figure 2.4 gives a clear picture of how the computation time is distributed in the OpenMP version of this application for the eta'K model we introduced in Chapter 1, evaluating 100 000 events. Since this model involves the computation of Gaussian functions (as most models do), a lot of time is spent calculating the exponential. Modern CPUs have vector units that are able to do one simple arithmetic operation on a set of data elements in true parallel (typically 128 bit, which means four floats or two doubles). The instructions used to operate these registers are called SSE (Streaming SIMD Extensions). Being able to utilize vectorized routines for e.g. exponentials is therefore of high importance, and Figure 2.4 shows that the compiler we are using (Intel C/C++ Compiler) is able to utilize the SVML (Short Vector Math Library) intrinsics that Intel provides, by the call

Function	Time
__svml_exp2.N	28.8%
PdfPolynomial::evaluateOpenMP	11.7%
PdfAdd::evaluateOpenMP	11.5%
PdfProd::evaluateOpenMP	11.0%
PdfGaussian::evaluateOpenMP	10.1%
PdfBifurGaussian::evaluateOpenMP	8.2%
PdfArgusBG::evaluateOpenMP	5.9%
[libiomp5.so]	4.4%
AbsPdf::GetVal	3.5%
[Import thunk __svml_exp2]	1.2%

Figure 2.4: An Intel Vtune Amplifier 2011 hotspot profile of the optimized OpenMP version of the MLFit application running with 100 000 events and 4 threads on an Intel Core i7 965 machine. Note that this is an excerpt of the contributing functions, showing the most time consuming ones.

to *__svml_exp2.N*. In this single case, there is not much we can do with the number #1 hotspot for the evaluation of this model, since it is an intrinsic function (which of course is a good sign). Note also that parallelization with OpenMP does not come without a cost. We can see that the OpenMP library (*libiomp5.so*) contributes directly with 4.4% of the total running time when running with 4 threads. This was a trivial example in the way that it is for a relatively low number of events. Chapter 3 provides more details regarding scalability and further optimizations.

GPU version There has also been developed a GPU version of MLFit, to explore what performance speedups modern GPUs can achieve. This is based on NVIDIA CUDA, and the work is explained thoroughly in [23]. Since each PDF class has a function *evaluateOpenMP*, switching to CUDA is not very difficult if one implements a corresponding function for the GPU. This function must transfer the necessary data onto an available GPU, and call a CUDA kernel responsible for computing the corresponding function. One problem with this setup (which in fact is the only effective setup possible) is that the number of GPU kernel calls is equal to the number of PDFs. Ideally, most of the time should be spent on computation, and therefore one should try to minimize the number of kernel calls (a GPU kernel call can be more expensive than a CPU function call). Because of this penalty, and because GPUs in general are throughput machines, a hypothesis is that the speedup of the GPU version compared to the CPU version will increase as the number of events increase (amortizing the overhead with more work). We will look further into the use of GPUs in Chapter 4.

2.5 Principles of parallel computation

We now introduce some fundamental concepts around parallel computation, which is essential in our terminology and in our reasoning about the execution of MLFit. A central point is to be able to scale the application across cores, since the trend within CPUs is to increase the number of cores, at least by the time this is written. This phenomenon is more thoroughly explained in Chapter 3.

2.5.1 Flynn's taxonomy

A classification of computers called Flynn's taxonomy describes four computer classes in both a serial and parallel context [24]. The four classes are:

- **SISD** - *Single instruction stream - single data stream.* A single processor computer that executes one stream of instructions on one set of data. Single-core processors belong in this class.
- **SIMD** - *Single instruction stream - multiple data stream.* A multiprocessor where each processing unit executes the same instruction stream as the others on its own set of data. In other words, a set of processors share the same control unit, and their execution differs only by the different data elements each processor operates on.
- **MISD** - *Multiple instruction stream - single data stream.* A multiprocessor where each processing element executes its own instructions, but operates on a common data set among all elements. This is a kind of conceptual class, since it is not widely used in practice. However, it can be found in bioinformatics for instance, where many processors operate on the same DNA string, issuing different types of instructions.
- **MIMD** - *Multiple instruction stream - multiple data stream.* A multiprocessor where each processing element executes its own instruction stream on its own set of data. This is the most usual class, where modern multi-core computers belong.

With SIMD we in practice understand computers that are specifically designed to execute a similar amount of instructions on different data elements. Vector computers fit to this description, as well as modern GPUs and also modern CPUs (partly) since they have vector registers constructed to issue one instruction on a set of data in the same cycle. However, when running a typical multi-threaded program it will primarily be regarded as a MIMD-style program, since the synchronization/order of execution is not deterministic. It is beneficial

to use the vector capabilities of modern hardware whenever possible, and much of our work will be centered around that.

2.5.2 Amdahl's law vs. Gustafsson's law

Exploiting every core on a modern multiprocessor to achieve hopefully large speedups is desirable. However, utilizing 8 processor cores instead of 1 does not mean an immediate speedup of $8x$. Parallelizing programs will often incur overheads like thread synchronization, and running more than one thread on a physical CPU (with more than one core) will in many cases lead to having to go further down in the memory hierarchy and eventually lead to loss of performance. However, there are one significant concept that can really limit the amount of speedup possible, and that concept is called *Amdahl's law.* A parallel program will in most cases contain a serial part, i.e. a piece of execution that has to be performed serially by one processing unit. Let t_s and p_s be the serial and parallel computation time, respectively. Speedup is then defined as

$$S(p) = \frac{t_s}{t_p}. \tag{2.5.1}$$

If we generalize the example with 8 processor cores above and denote p as the number of processors, the maximum speedup of a parallelization would be if $t_p = \frac{t_s}{p}$ and thus $S(p) = \frac{t_s}{\frac{t_s}{p}} = p$. As we mentioned above, t_p will in most cases involve serial parts of execution. Let f denote the fraction of the computation that is serial. We could then rewrite equation 2.5.1 into

$$S(p) = \frac{t_s}{ft_s + (1-f)t_p} = \frac{t_s}{ft_s + \frac{(1-f)t_s}{p}}. \tag{2.5.2}$$

We are interested in finding the maximum speedup when the parallel computation involves serial parts. If we now let p grow, we see that

$$S_{max} = \lim_{p\to\infty} \frac{t_s}{ft_s + \frac{(1-f)t_s}{p}} = \frac{t_s}{ft_s} = \frac{1}{f}. \tag{2.5.3}$$

This means that the maximum speedup is strictly limited by the serial fraction. If a parallel application involves a serial fraction of 5%, the maximum speedup would be $\frac{1}{0.05} = 20$, independent of how many processors one throws at the job. It is a lot easier to see the direct effects of this with a plot, which is illustrated in Figure 2.5.

A prerequisite for Amdahl's law to apply is that the problem size, N, is fixed. This is also

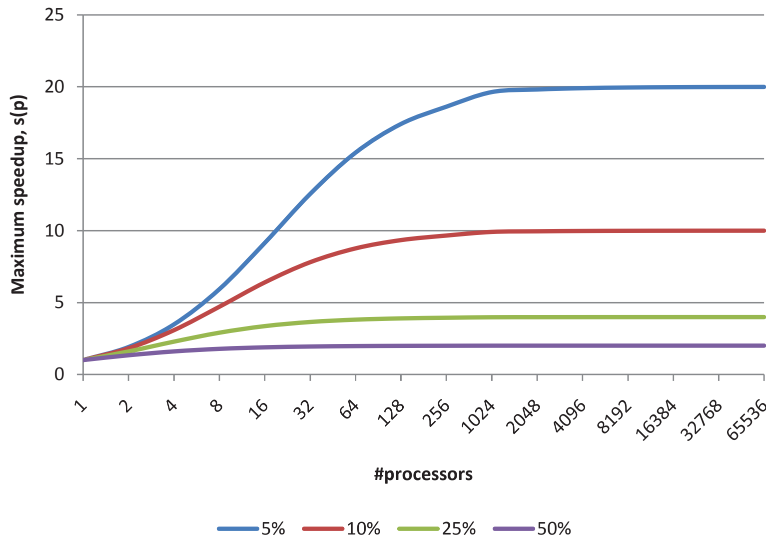


Figure 2.5: An illustration of Amdahl's law for a parallel computation with serial fractions of respectively 5%, 10%, 25% and 50%.

called *strong scaling* and could be summarized as "how well does an application scale by the number of processing elements when the problem size is fixed". However, if one changes the problem size as one changes the number of processing elements, and assumes that the serial fraction is not dependent on N, the time spent by the serial part will be amortized. This is based on that as more and more compute units are used, one is able to compute a much larger problem set in a time comparable to the serial one. This is called Gustafsson's law, or *weak scaling* and can be summarized as "how well does an application scale by the number of processing elements when the problem size per processing element is fixed, and the serial part is not dependent on the problem size".

In our case, the serial fraction consists of a serial reduction of partial parallel-reduced values (one partial sum per thread), which means that the serial fraction is only (weakly) dependent of the number of threads. This is absolutely ideal.

2.6 Numerical sensitivity

As stated in Section 2.2, MLFit uses the MINUIT package for performing the function minimization itself. However, by experimentation, MINUIT is very sensitive regarding the numerical result of the *NLL* function, which is understandable since parameter space search can involve consecutive results that are very similar in value. A minor numerical error can

lead the minimization algorithm on a wrong path from the start, which is not good. This has some consequences regarding the optimization efforts. The OpenMP version uses a parallel reduction to compute the sums of the negative terms in the *NLL* equation (equation 2.2.4). However, the order of the summation must be preserved between each call of the *NLL* function for MINUIT to work properly, since a summation of different order would lead to a different result (non-identical bit-level value). This is because the law of associativity breaks down since rounding is applied in floating-point arithmetic, i.e. it *might* be the case that

$$(u \oplus v) \oplus w \neq u \oplus (v \oplus w). \tag{2.6.1}$$

The reason for this is that the relative error for a pairwise addition of any of u, v and w can differ depending on what the operands are, since each operand (u, v or w) have different relative errors compared to the exact decimal value [25]. The parallel reduction has therefore been written to conform with these requirements. It preserves the order of the operations for a given number of threads and it reduces the rounding problem due to associative floating point arithmetics using the double-double compensation algorithm 2Sum [26]. In this way the results are deterministic and stable in all tests. However, if a user runs the application with i number of threads and produces a result, he can not be sure to get a bit-identical result if he runs the application again with j number of threads afterwards (given $i \neq j$), since the order of summation definitively would have changed.

MLFit is written using double precision floating-point numbers exclusively, and we do not know if we are able to use single precision floating-point numbers. A trend among physicists is to use double precision "to be sure", but single precision is often enough for many kinds of analysis. If we could use single precision in the evaluation, it could mean a potential $\sim 2x$ speedup on CPUs (doubling the vector registry capacities), and even more on GPUs (GPUs were originally made for single precision calculations, and they have special features geared towards that). MINUIT uses double precision exclusively, and we do not know how MINUIT would react if we return single precision values to it. Following the IEEE-754 standard [27], single precision has a fractional part of the mantissa represented by 24 bits. Therefore the highest accuracy representable converted to decimal form is $log_{10}(2^{-24}) = 7.2247...$, which gives us a precision of 7 digits. If MLFit uses single precision exclusively, it may be that MINUIT will not be able to see the difference between value NLL_i and value NLL_{i+1} if they differ *only* after the 7th digit, since we sacrifize precision. The effects of this problem could be catastrophic for the minimization.

In addition, it should be mentioned that MINUIT is not the only minimization algorithm

out there. Another well-known approach for function minimization is the use of genetic algorithms [28], which could possibly work fine with single precision, but this is an assumption. We focus our work around how much speedup it is possible to get when doing *NLL* evaluations with this package on modern hardware, since precision requirements differ from algorithm to algorithm.

2.7 Eta'K model description

In the eta'K model there are 3 observables (variables) and 5 species. In total there are 29 PDFs. All PDFs have an analytical integral, so no numerical approximations of integrals must be done. The C++ implementation of this model is given in Appendix C, and a tree illustration of it is given in Appendix D

Chapter 3

MLFit on multi-core processors

This chapter describes what we have done to improve the multicore version of MLFit. We explore other technologies that supports multi-core parallelism, and we perform general optimizations. An important feature of MLFit is how programmable it is, and solutions that are easily modifiable as well as delivering good performance/scalability will be preferred. With that in mind, using a technology that could support CPUs and also GPUs could maybe be interesting. Since RooFit is often run by physicists on their own computers, commodity hardware is our main target segment and therefore our tests are run on such hardware.

3.1 The motivation for multi-core processors

After year 2000, it became more and more usual to produce and sell multi-core processors. Many computer users are using machines made for parallel execution often without knowing it. Moore predicted in 1965 that the number of transistors possible to place on a single chip would double every 2 years, and there is currently no sign that this trend will stop [29]. For many years, processor manufacturers made the processors faster and faster by increasing their frequency (clock rate), but this came to a stop because of physical constraints. A.R. Brodtkorb et al. mentions in [29] the formula for *power density* (or *power wall*),

$$P = C\rho f V_{dd}^2, \tag{3.1.1}$$

where P is the power density in Watts per unit area, C is the total capacitance, ρ is the transistor density, f is the processor frequency and V_{dd} the supply voltage. Higher voltages increase the potential processor frequency, and it is easy to see by the formula that the power density will reach extremely high values if continuing to push the frequency and voltage.

With large power consumption comes large temperatures, and with a growth proportional to the square of the voltage, power consumption will in the end become enormous and demanding cooling techniques have to be applied. To solve this problem, (i.e. to keep P constant) processor manufacturers have temporarily chosen the strategy of duplicating the processor cores (increasing transistor density) instead of increasing their frequency, called SMP (symmetric multi-processing). In an SMP processor, a number of processing cores reside on one physical chip, often sharing the last-level cache (L3). According to [29], the rule of thumb interpretation of 3.1.1 is that if one decreases the voltage and frequency by 1%, the power density is decreased by 3%, and the performance by 0.66%. Therefore will for instance dual-core processors which are running at 85% of the frequency with 85% of the supply voltage have the possibility to offer 180% better performance than an equivalent single-core processor. This is of course an elegant solution to the problem, but it imposes challenges. Programming parallel applications is more time-consuming than programming serial ones, and many programmers are not trained in parallel programming. It can also often be challenging to achieve linear speedup with respect to the number of cores you have, since most programs contain serial parts (albeit often small ones), and it is not unusual to experience overhead regarded to threading mechanisms/libraries. Also, since the L3 cache on SMP processors is often shared between the cores, it is important to take special care when accessing memory to achieve a good cache hit rate when increasing the number of threads. If you wanted a speed boost in the single-core era you could buy a faster processor with higher frequency. Today, you can not be sure that your favourite program supports multithreading, so buying a brand new multi-core processor will not necessarily increase performance. Actually, since the frequency might be reduced, you might experience worse performance.

MLFit is already parallelized and takes advantage of all the cores you want (easily adjustable by OpenMP). But are there libraries than can to the multithreading job better, or are the most elementary threading techniques the best for this application? We try to answer that question in the following sections. We have tried different technologies because we wanted to get a picture of the possibilites programmers have. We believe this can also reveal interesting results for other groups at CERN heavily involved in parallelization of software.

3.2 Alternative threading technologies

In this section we compare the performance and programmability of two other threading technologies to see if they in our case are good alternatives to the OpenMP industry standard.

3.2.1 Brief OpenCL introduction

OpenCL is a standard which defines an API for programming heterogeneous computers, created and maintained by the Khronos Group, which is a group of industry vendors that have interest in promoting a general way to target different computational devices [30]. It abstracts a parallel execution with a concept called a *kernel* (or *computational kernel*). This kernel is written on a *per-thread* basis, and the runtime executes this kernel by laying out threads grouped in independent groups called a *workgroup* on a thread *grid*. Each work group consists of a given number of threads, and each thread is called a *work item*. The main benefit with such a structure is that it maps well onto SIMD algorithms, and thus is ideal for implementing computations to be performed on a GPU. This is first and foremost because threads on modern GPUs are lightweight and can be spawned and scheduled with much less overhead than on a CPU [31]. Workgroups can again be divided in to small groups of threads called *wavefronts.* We will revisit these in Chapter 4. We show an illustration of this thread grid in Figure 3.1, which is a reprint from [31]. This concept will become more clear in the next section where we provide some code samples for this paradigm of programming.

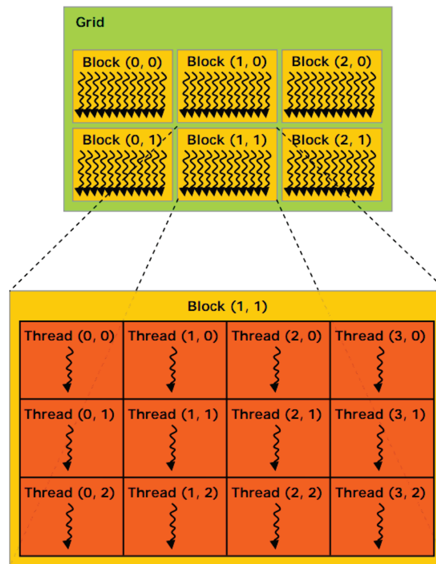


Figure 3.1: An OpenCL thread grid consisting of workgroups (or *blocks* in NVIDIA terminology).

3.2.2 OpenCL for CPUs

MLFit has been implemented with OpenMP, but it is interesting to explore if OpenCL and its programming paradigm would be suitable for it too, in addition to delivering competitive performance. Ideally, it should be possible to run the same code on both CPUs and GPUs while still achieving good performance on both platforms, but that remains to show.

Listing 3.1: OpenMP version of a Gaussian evaluation function.

```
void evaluate(const double mu, const double sigma, const double* data,  double* results,
    const int N)
{
  #pragma omp parallel for
  for(int i = 0; i < N; i++)
  {
    double temp = (data[i]-mu)/sigma;
    temp *= temp;
    results[i] = exp(-0.5*temp);
  }
}
```

Listing 3.2: The OpenCL equivalent of the function in listing 3.1.

```
 __kernel void evaluate(__const double mu, __const double sigma, __global  const double *
     data, __global double *results, __const int N)
{
  int i = get_global_id(0);
  if (i >= N) return;
  double x = data[i];
  double temp = (x-mu)/sigma;
  temp *= temp;
  results[i] = exp(-0.5*temp);
}
```

Implementation

Listing 3.1 shows an OpenMP version of a function which evaluates a Gaussian function. Listing 3.2 shows the equivalent OpenCL kernel. Each thread that is configured to run this kernel, will query the OpenCL runtime for its ID. If the ID is in the scope of the number of elements in the array, it will compute the function evaluation for one single element. The programming model therefore assumes that thread scheduling is implemented. If N is higher

than the number of threads, which would almost exclusively be the case, it is completely necessary to have effective scheduling of a large numer of threads to make the execution effective. This is the case for GPUs, and the programming model is therefore highly suitable for GPUs.

AMD has released an OpenCL implementation supporting x86-compatible CPUs, which is available for Windows and Linux. The name of the package is AMD APP SDK, and version 2.3 was released in January 2011 [32]. It should be noted that Intel did not release an OpenCL implementation for Linux before the end of the period of this thesis. We want to see if AMD APP SDK is comparable performance-wise to Intel's implementation of the mature industry standard OpenMP technology. The OpenMP version of the MLFit application implements parallelism by a function called *evaluateOpenMP* in the class RooAbsPdf. Each inheriting class then implements this method itself which returns the result array of doubles for the whole N-range. The high function-level coherence makes it easy to swap this function with a function called *evaluateOpenCL*. However, there are some differences between the two technologies.

OpenCL is all about generality, and this means that one has to treat CPUs on the same device level as GPUs and other compute devices. In GPU applications, it is necessary to transfer data from the host's main memory to the GPU. Since CPUs are also treated as devices, it is in plain CPU OpenCL applications necessary to transfer data "to the CPU", but this only means either copying from one area in memory to another (which would be slow and unnecessary), or using the data at the source address (faster and more elegant). This is implementation specific, and we have to trust the vendors in making an effective implementation. This copy operation makes the code more verbose, and it is necessary to work with structs of *cl_mem*, which represents memory objects (by our experience, essentially a wrapper over a pointer to memory, but this is of course also implementation specific) in the OpenCL runtime. Also, when invoking kernels, it is necessary to invoke functions to set kernel arguments, and this has to be done with one call per argument. Listing 3.3 illustrates this for the call to the Gaussian evaluation kernel. This is clearly very verbose, and could almost be regarded as code bloat, atleast when comparing to the code for the OpenMP version. When the kernels grow in numbers, this increases the amount of code significantly.

The OpenCL runtime compiles kernel files at runtime, and turns the executable equivalents into binaries represented in code by *cl_kernel* structs. This way, all functions and constants the kernels use, has to be in the source compiled by the OpenCL compiler. Unfortunately, in our case (RooFit), many functions (mathematical PDFs for example) are already implemented and tested in C++. This implies duplication of code if we want to support these

functions in OpenCL, since it is not possible to do procedure calls from the OpenCL environment to C++ code compiled by the C/C++ compiler. Anyways, since we aim to implement GPU support by OpenCL too, this is of lesser significance (the functions must be ported anyways).

Listing 3.3: A wrapper function for calling the OpenCL Gaussian evaluation kernel in listing 3.2.

```
void OpenCL::evaluateGaussian(const Double_t mu, const Double_t sigma, cl_mem data, const
     int variableOffset, cl_mem results, const UInt_t N)
{
  size_t totalSize = N;
  clSetKernelArg(OpenCL::m_evaluateGaussian, 0, sizeof(Double_t), (void*)&mu);
  clSetKernelArg(OpenCL::m_evaluateGaussian, 1, sizeof(Double_t), (void*)&sigma);
  clSetKernelArg(OpenCL::m_evaluateGaussian, 2, sizeof(cl_mem), (void*)&data);
  clSetKernelArg(OpenCL::m_evaluateGaussian, 3, sizeof(cl_mem), (void*)&results);
  clSetKernelArg(OpenCL::m_evaluateGaussian, 4, sizeof(UInt_t), (void*)&N);
  cl_int status = clEnqueueNDRangeKernel(OpenCL::m_queue, OpenCL::m_evaluateGaussian, 1,
      NULL, &totalSize, NULL, 0, NULL, NULL);
  CL_CHECK_ERROR(status, "evaluateGaussian");
}
```

The arguments to the *clEnqueueNDRangeKernel* function includes parameters that describes the thread grid that is set up. The variable *totalSize* represents the number of elements in the grid. If one wants to configure the execution in a more detailed way, it is possible to pass the number of work items per workgroup also (local size). In this example we pass the value *NULL* which makes the OpenCL implementation decide the workgroup size. This could be a decent choice when running this on a GPU, but as we will see soon, not for a CPU (using AMD APP SDK).

Drawbacks and optimizations

Threads on modern GPUs are very lightweight, and scheduled by hardware mechanisms (we describe GPUs more carefully in Chapter 4). In an application performing calculations on e.g. vectors, it is therefore appropriate to make each thread typically target one single element in that vector. If the number of threads is smaller than the number of elements, threads that are done with their execution will be rescheduled to compute another element without much scheduling overhead. This is automatically taken care of by the OpenCL library/driver, which clearly eases the programming effort. It is tempting to use one unified programming model for a range of devices, however, using the OpenCL implementation in AMD APP

SDK for the CPU is not necessarily straightforward if one wants to achieve performant code. Using a kernel implementation strategy similiar to the one in listing 3.2 fits GPUs well, and it would be highly ideal if this kernel could be highly performant also on the CPU. However, this is not the case for two main reasons: AMD APP SDK does no auto-vectorization (which is done by the Intel C++ compiler in the OpenMP version) and the way to do threading on the CPU is different from the GPU.

Vectorization Since the OpenCL compiler in the AMD APP SDK does not support auto-vectorization, one has to explicitly program with vector types defined in the OpenCL standard to make the compiler produce vectorized x86 code (SSE). By introducing explicit vectorization in the code we now have a different kernel, and the benefit of a unified programming model is dramatically reduced, since we need one version of the same kernel for both CPUs and GPUs. It would be a lot easier if the AMD APP SDK OpenCL compiler had auto-vectorization capabilities. Listing 3.4 shows the vectorized edition of the kernel in listing 3.2. A positive side with this OpenCL implementation is that the syntax for loading and storing vector types is clean and easy to understand, and vector types have mathematical operators implemented.

Listing 3.4: The vectorized version of the kernel in listing 3.2.

```
__kernel __attribute__((vec_type_hint(double2))) void evaluatePdfGaussian(__const double
    mu, __const double sigma, __global const double *data, __global double *results,
    __const int N)
{
  int i = get_global_id(0);
  if (i >= N/2) return;
  double2 x = vload2(i, data);
  double2 temp = (x-mu)/sigma;
  temp *= temp;
  double2 result = exp(-0.5*temp);
  vstore2(result, i, results);
}
```

Note that since it is necessary to program this by hand, we must have arrays of even length the way the source code is now, i.e. it is necessary to take special care off odd elements. Also, if we want to use larger vector registers in the future, e.g. 256-bit AVX (Intel Advanced Vector Extentions [33]), we have to change all the vector types in the code. This could be possible to bypass with a type definition for vector types, but in general, this should be done by the compiler.

A major drawback of AMD APP SDK is that it does not support vectorization of exponential functions. As we have already seen, the Intel compiler, with optimizations turned on, compiles exponential functions down to calls to the SVML vectorized exponential routine, but this is not the case for AMD APP SDK. This is a *major* drawback, and suggests that an OpenMP implementation compiled with a vectorizing compiler is much more suitable since functions like the exponential often ends up being one of the major hotspots in our case.

Thread scheduling With the AMD APP SDK, the kernel in listing 3.4 will not perform very well on a CPU, even though vectorized code is emitted (which it to just some degree is). This has to do with how the AMD APP SDK handles thread scheduling internally. We do not know the details of the implementation, but AMD encourage users (in online examples on their App SDK web pages [32]) to give CPU threads more work than GPU threads. Tests we have conducted, showed that running CPU kernels as the one in listing 3.4 resulted in a performance of around 30% compared to an auto-vectorized OpenMP version. However, by splitting the data into an appropriate amount for each thread, we achieved speeds similar to it (scaled with respect to the lack of vectorization of transcendental functions).

Listing 3.5: Same as listing 3.4, but with more work per thread.

```
__kernel __attribute__((vec_type_hint(double2))) void evaluatePdfGaussian(__const double
    mu, __const double sigma, __global const double *data, __global double *results,
    __const int N, __const int numComputeElements)
{
  int i = get_global_id(0);
  if (i >= N) return;
  int part = N/numComputeElements;
  for(int index = i*part; index < (i+1)*part - 1; index+=2)
  {
    double2 x = vload2(index/2, data);
    double2 temp = (x-mu)/sigma;
    temp *= temp;
    double2 result = exp(-0.5*temp);
    vstore2(result, index/2, results);
  }
}
```

Listing 3.5 shows a kernel that does more work per thread. Note that this kernel assumes that the number of compute elements evenly divides N. Doing more work per thread forces the developer to think about work distribution, and then most of the benefits with this programming model are lost, since it really should be implicit. Work distribution is not very difficult to achieve, but it would be like fighting against the programming model. The ideal case would be to have the same kernel for both the CPU and the GPU, and that the OpenCL SDK took care of this automatically. As we can clearly see, the kernels in listing 3.2 and 3.5 are very different from each other. Also, note that the Gaussian function is a trivial function to implement. Other functions might be much more complex.

3.2.3 Preliminary OpenCL conclusion

We do not see any reason to port the evaluation functions in the CPU version of MLFit from C++ to OpenCL, atleast not for now. OpenCL for CPUs has to mature a lot before this can be attractive. We want the compiler to deal with vectorization, and we want a threading library that can do effective scheduling of threads, without forcing the programmer to do programming tweaks. The primary argument for using OpenCL is portability. We could for each function use one OpenCL kernel for both CPUs and GPUs, but that would lead to performance penalties. For the performance penalties to become smaller, one would have to have separate OpenCL kernels for each device, and then there is really no point in

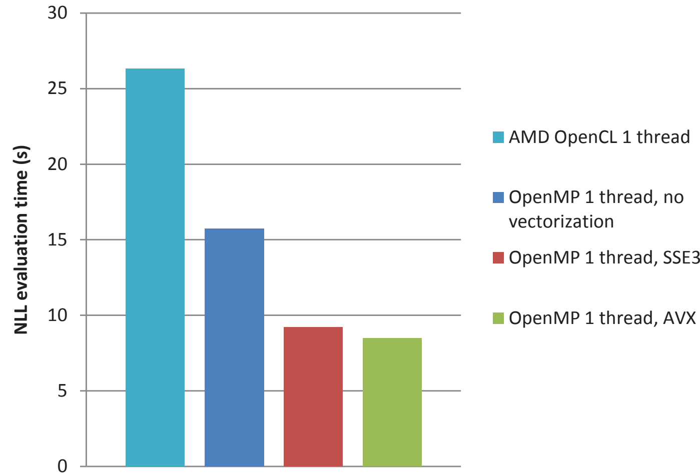


Figure 3.2: Performance comparison of a single-threaded evaluation of the eta'K model for OpenCL versus OpenMP with the Intel C compiler both with and without vectorization (both SSE and AVX). 1 000 000 events.

implementing OpenCL kernels for the CPU if one already has a well established C++ version with OpenMP. Figure 3.2 shows a comparison for a single-threaded run of an evaluation of the eta'K model with 1 000 000 events. Clearly, AMD's OpenCL implementation is not close to near the non-vectorized OpenMP implementation compiled by the Intel C compiler because of unnecessary overhead in the library. We emphasize that the OpenCL version (i.e. the C++ part of the OpenCL version) also was compiled with the Intel C compiler.

We have in addition tried Intel's OpenCL version (a Linux edition was released at the end of the period of this thesis) as a preliminary evaluation for Intel, without much success regarding performance. Intel's implementation was comparable to AMD APP SDK.

3.2.4 Intel Threading Building Blocks

Intel Threading Building Blocks, or TBB for short, is a multithreading library developed by Intel. It is open source, written in C++, and aims to give programmers a way to introduce multithreaded data and task parallelism in C++ programs. MLFit is mainly data parallel (for now), so the tools for managing tasks by explicitly using task constructs in TBB is not interesting for us at this moment. We will use one TBB method primary, and that is the templated *parallel_for* method. Moving to TBB is a relative straight-forward operation. Since parallelism is encapsulated in a single method of each PDF, this is the main piece of code we have to change.

Listing 3.6: C++/pseudocode describing the use of the TBB *parallel_for* method.

```
//Prototype of parallel_for
void parallel_for(blocked_range<size_t>, <functor>, tbb::partitioner);
//An accumulation functor
class PdfGaussianTBBAccumulator
{
  PdfGaussian* m_pdf;
  Double_t* m_data; public:
  void operator()( const tbb::blocked_range<size_t>& r) const
  {
    int end = r.end();
    for(size_t i = r.begin(); i < end; ++i)
    {
      /*
      * Calculate the gaussian for element i based on the data array,
      * and put the result in some result structure in m_pdf
      */
    }
  }
  PdfGaussianTBBAccumulator(PdfGaussian* pdf, Double_t* data) : m_pdf(pdf), m_data(data)
      {} };
//Executing the parallel evaluation of the Gaussian function
parallel_for(blocked_range<size_t>(0, N), PdfGaussianTBBAccumulator(pdf, data), tbb::
    auto_partitioner())
```

Listing 3.6 shows some code which explains how execution with *parallel_for* is done. The method accepts an object called *blocked_range*, which is a TBB object representing a partition of the array. Since TBB is based on tasks, and that threads take/steal tasks, each thread will take a task representing a blocked range and perform the computations in this area. If the thread is done and there is still more work to be done, it will be scheduled to take another range. The explicit declaration of the integer *end* is done to help the compiler vectorize the loop.

For partitioning the work, TBB offers different work partitioner implementations. A standard partitioner is the *auto_partitioner* used in listing 3.6. It uses a heuristic to decide the range size for each task. Note that it is also possible to define a static blocked range size (also called *grain size*). Another type of partitioner is the *affinity_partitioner*, which we assume is a partitioner optimized for good cache usage. All our tests has shown slightly performance improvements using this partitioner. It is in general difficult for us to describe it more detailed than this, since this is like a black box. We show results including TBB runs in

Section 3.4. An important aspect of TBB is to provide enough work for each thread, so that the overhead of calling *parallel_for* is justified/amortized. According to [34], it is suggested that each thread has a grainsize so large that the execution of *parallel_for* should take at least 10 000 to 100 000 instructions to execute for each thread. This can potentially involve a penalty for low workloads, and we are looking for a solution with maximum performance in any case, since we cannot assume what N will be.

Downsides TBB is from a programmer's perspective an elegant tool to use for parallelism. However, the programmer cannot identify a logical thread, i.e. the OpenMP phenomenon of thread IDs does not exist. Actually, this is the purpose of TBB. The programmer should not have to deal with threads and do for instance SPMD[1] branching with respeect to thread ID. The idea is that the programmer should map tasks onto work, and not care about which threads execute those tasks. Unfortunately, this is problematic since MINUIT requires a deterministic reduction for each evaluation, as mentioned in Section 2.6. To achieve this, one has to explicitly specify which elements each thread will reduce. But this is totally in opposition to what TBB is made for, and as mentioned, we cannot use thread IDs to assert a deterministic reduction.

TBB also requires implementation of more code to work. It is necessary to implement one class per *parallel_for* operator, which is not elegant compared to OpenMP. Another point is that TBB is a stand-alone library. OpenMP is included in most compilers on the market, which obviously is easier for users.

3.3 General optimizations

This section describes the general optimizations we have done to MLFit. All these optimizations comes on top of the optimizations already done to it, presented in Section 2.4.

3.3.1 A different evaluation approach

The optimized/parallelized OpenMP *NLL* evaluation in MLFit is implemented as follows; A call to *GetVal* in a leaf PDF will start an OpenMP parallel region, and call the method *evaluateOpenMP* which will vary by polymorphism with respect to which class one is inside.

[1] *SMPD* is a term used for a MIMD execution with a number of processes/threads starting running the same instructions, but later executing an individual set of instructions with the help of branching inside the program. This is often used in e.g. MPI programs.

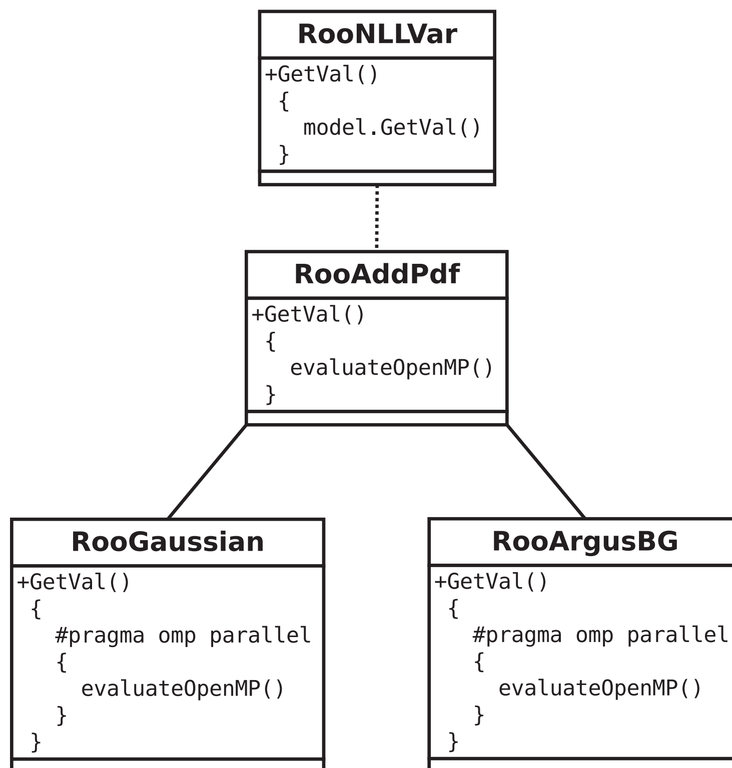


Figure 3.3: An explicitly parallel evaluation of a PDF tree.

Figure 3.3 shows a parallel version of the scenario in Figure 2.3. When entering the *GetVal* function of RooAddPdf, a parallel region will *not* be started, since we are in a composite node. This function will single-threadedly call its own *evaluateOpenMP* function, which for RooAddPdf is specifically implemented to iterate over all child PDFs and call their *GetVal* function too, traversing the tree, for in the end to sum these partial results. However, each *GetVal* function in leaf nodes (RooGaussian and RooArgusBG in this case) applies the actual parallelization, and this means a new OpenMP parallel region which we suspect could involve a larger overhead than necessary. We would want as few OpenMP parallel regions as possible, since threading overhead should be kept at a minimum. To remove this potential overhead, we have rewritten the entire evaluation using a different pattern. The idea is to have one parallel region only, and this region will be started at the top of the tree. To avoid race conditions in the composite PDFs, we statically partition the problem set (which was also done before), and make each thread do an evaluation from the root to the leaves within its own partition only. We call this an *implicitly parallel* approach, since the parallel execution will be defined another place than in the PDFs, and now does not have to stop at various places inside the tree, but can be run on the entire tree in a direct top-down manner,

avoiding eventual thread overhead as much as possible. In practice, this is implemented as a parallel OpenMP region in the class responsible for doing the *NLL* evaluation. This way, no PDF classes need to know anything about OpenMP since the parallel call happens only in the *RooNLLVar* object. This means that parallelization is much easier, and that simpler threading mechanisms eventually can be used with very little implementation work. In other words, we do not really need any sofisticated threading technology, since this is a statically balanced evaluation with one entry point for parallel execution. This is very beneficial with respect to the fact that the implementation is loosely coupled from any threading technology, and we have reduced the amount of code needed to do threading to a few lines.

This evaluation pattern is not an optimization without consequences that may be problematic. When evaluating the PDF evaluation functions in parallel without an explicitly parallel region inside them, it is crucial that these functions do not modify member variables of the object the method is run on, or global variables, without carefully assuring that race conditions are avoided. An illustration of the implicitly parallel evaluation is shown in Figure 3.4.

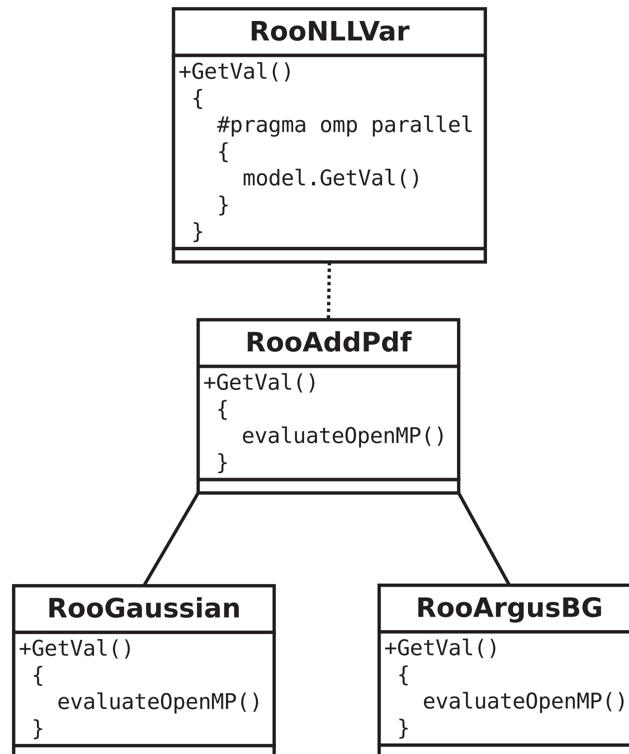


Figure 3.4: An implicitly parallel evaluation of a PDF tree.

3.3.2 Aiming for scalability

Scalability characteristics changes depending on which hardware an application is run on. Some processors might have loads of cache, while smaller commodity systems have smaller cache sizes. Scalability is a problem for MLFit on commodity hardware, and this is not a problem of Amdahl's law since we see this trend on large numbers of N only. Just to illustrate, we have in Figure 3.5 included two VTune hotspot profiles from the implicitly parallel version running 4 threads with 100 000 and 1 000 000 events respectively. These show that there is clearly problems inside composite PDFs when N grows large.

/Function /Call Stack	CPU Time
__svml_exp2.N	39.1%
PdfPolynomial::evaluateOpenMP	11.5%
PdfArgusBG::evaluateOpenMP	8.2%
PdfGaussian::evaluateOpenMP	6.8%
PdfAdd::evaluateOpenMP	6.5%
[libiomp5.so]	6.1%
PdfProd::evaluateOpenMP	5.4%
AbsPdf::GetVal	4.3%
NLL::GetVal	3.6%
PdfBifurGaussian::evaluateOpenMP	2.9%

(a) N= 100 000

/Function /Call Stack	CPU Time
PdfProd::evaluateOpenMP	22.8%
__svml_exp2.N	18.5%
PdfAdd::evaluateOpenMP	17.8%
PdfPolynomial::evaluateOpenMP	11.8%
PdfGaussian::evaluateOpenMP	6.6%
AbsPdf::GetVal	6.6%
PdfBifurGaussian::evaluateOpenMP	4.7%
PdfArgusBG::evaluateOpenMP	4.4%
[libiomp5.so]	2.1%
__svml_log2.L	1.7%

(b) N= 1 000 000

Figure 3.5: VTune hotspot profiles for the OpenMP implicitly parallel version on 4 threads. N is 100 000 and 1 000 000 respectively.

The relative time spent on actual computation is here reduced dramatically (can be seen from the time used on doing exponentials in SVML). In composite PDFs, it is necessary to evaluate the child PDFs and then add/multiply PDF by PDF. We have verified that the local hotspots inside the prod and add functions correspond to the source lines that do the loading of child results from memory, as well as storing the final result to memory. This incurs a high load on caches, and will for a large number of events lead to a lot of L3 cache misses. This can be explained by the fact that doing an addition or a multiplication is almost negligible compared to several memory operations, while in the evaluation functions of the PDFs, transcendental functions increase the relative time spent on actual computation (i.e. the *arithmetic intensity*). Note that it seems like the polynomial PDF is relatively expensive compared to the other PDFs. This is because the exponential routine is separated from the PDFs, so the Gaussian function for instance spends more time than depicted by the profile if one adds the time spent in the exponential routine.

A common technique used to solve this problem is to do *cache blocking*, also called *block*

splitting. Cache blocking works by splitting the data domain into blocks and consuming one by one of them. This will hopefully increase locality and thereby cache efficiency. There were two difficult areas to consider when doing this optimization;

- When doing cache blocking in e.g. a for loop, and with more than one thread, it is necessary to compute which index each thread should access by using which block to target, the thread's ID and the loop counter. It is not always possible for the compiler to know if the indices produced by these expressions are aligned or not, since they often are non-deterministic at compile time. This will again lead to that the compiler eventually cannot vectorize the loop, which implies a huge performance degradation since SSE instructions are not emitted.

- If cache blocking is to be used, it means evaluating the total function the tree represents in chunks, and thereby more call to virtual functions than doing the whole evaluation in one. This can also be of large significance, especially if the model (tree) consists of many PDFs, since each node represents a virtual function call for each iteration. All in all, this is a trade-off situation that is difficult to optimize, at least when it is not known a priori which hardware the application is run on.

The solution that gave improvements was actually implemented in the old OpenMP version (explicitly parallel) before this work was started. It was tried as an optimization, but that was in a compute-bound scenario (larger machine) which then did not pay off. However, in our commodity scenario the application obviously is memory-bound for large numbers of N, and we have therefore implemented it also in the implicitly parallel version.

The solution is to run the *evaluate* function of the tree root on a block basis (in the *NLL::GetVal* function) and not do any changes inside the tree. When one thinks about it, setting a block size of e.g. 10 000, will be *almost* like evaluating the function for $N =$ 10 000, only that this happens many times, and by Figure 3.5, this suggests a much faster computation. Sometimes the simplest solution clearly is the best. By *almost*, we mean that there is one difference; when MLFit is run without block splitting, a static partitioning/load balancing will occur, and each thread will get one partition of the entire workload. When block splitting is applied, the exact same thing is done, but each thread's partition is blocked. Another blocking solution would have been to block the entire array and then partition each block into thread parts. The reason the former was chosen is that it is necessary to have partitions of fixed size to have a deterministic reduction. If the block size was to be changed (for performance tuning for instance) this would imply different results, since the partition

distribution would have changed. This would have meant a reduction value depending on the number of blocks and threads. However, when partitioning is done first, it is clear that each thread has n_i elements (where i is the thread ID) and that those elements can be consumed on a block basis, but will still be reduced to the same bitwise number independent of the block size (given that the number of threads is constant, of course).

3.3.3 Result propagation and loop fusion

Loop fusion (or *loop jamming*) is a technique where two or more subsequent loops are merged into one loop. The normalization step after the function evaluation means in practice to divide the function value by the integral of the function. In RooFit this was done by doing a division for each element, but when MLFit was written, and all values of a whole range were to be computed, it was then possible to multiply by the reciprocal of the integral instead (division is an inherently slow operation [35]). This optimization contributed a lot, since this happens for each value, for each PDF in the tree. The normalization, at the time being, happens in a separate loop which is run after the PDF evaluation. It is possible to merge this loop into the function evaluation loop, and we have done exactly that. The idea is not to just escape the loop overhead, but also to overlap computation and memory accesses. This can be seen as changing the design in RooFit, but it should be possible to keep this as a special function which does evaluation and normalization in one, while keeping the evaluation and normalization function separate (performance can still be kept because of function inlining).

Since there is a general hotspot in composite PDFs when N grows large, we have tried to come up with solutions that can lower the load on memory that these areas incur. As already described, the composite PDFs tell their children to produce their results, and then do the given operation (e.g. addition or multiplication) on those results, PDF by PDF (after the childrens' computation). But what if child by child could do this operation at the same time as they are calculating themselves, and thereby operate only on the parent result? This would mean that each leaf PDF does not have to store its own results anymore, and quite a few memory accesses are escaped. We have implemented a solution where each composite PDF sends its result array "down" to the child, so that the child can compute itself and also do the parent's operation on the parent's result. This is for sure a loop fusion, as well as a bottom-up propagation of computed results, and another positive bi-effect could be latency hiding since the computation is done in a memory-hotspot as opposed to before. This is possible since the evaluation of different PDFs is sequential within each thread. Also, since we are now operating on the parent results, a lot of memory is saved.

To summarize, a child PDF hereby has the responsibility to

- Evaluate itself.
- Normalize itself by a multiplication of the reciprocal of its integral.
- Perform the required operation that the parent represents on the parent result array directly, using its newly computed result.

This might sound as more work for programmers/users that maybe want to add new PDFs, and that is true to some degree. For each PDF it is necessary to add one function for each composite PDF type. In MLFit, this means two functions (add and prod) for now. It is not possible to implement it in another way in C++, since virtual functions cannot be e.g. templated, and we have to avoid branching and virtual function calls inside the evaluation loops, since that would break vectorization. It is clear that we introduce duplication of code to make the program run faster. This is something that must be considered when this prototype is going to be merged into RooFit, since it is a trade-off situation between good coding practice and performance.

3.3.4 Constant expressions

In Section 2.4 we described the changes done to the implementation when parallelizing it. In RooFit, there are many types of PDFs, and each PDF often has some unique variables which are used in the evaluation. To illustrate, we list the implementation of the *BifurGaussian* PDF in listing 3.7, which is called from its *evaluateOpenMP* function. This PDF relies on a coefficient in the evaluation, and this coefficient is calculated by doing a division and a multiplication of some other variables. This is maybe a rational implementation in RooFit where each evaluation returns *one* value, since you might want to change for instance *sigmaL* in between two function evaluations. However, when evaluating a whole range of values, it is desirable to keep *sigmaL* constant (but of course have the possibility to change it between the parallel evaluations), which means that this value could be precomputed before entering the computational kernel. This will lower the amount of arithmetic operations, and the performance gain will be proportional to the complexity of the model, i.e. one can benefit more if one has 5 *BifurGaussian* PDFs in the model than 1. This is not an architectural optimization, since it is more at the detail level, but we believe it could have an impact and thus should be mentioned here. The eta'K model consists of 4 BifurGaussian PDFs and 3 ArgusBG PDFs among others, and both of these are targets for constant expression

optimizations. But there are also PDFs that are not a target for this optimization. For instance, the gaussian function only relies on *x, sigma* and *mean* so it is not possible to compute any constant expressions in that case.

Making expressions constant is primarily the work of the compiler, and modern compilers do this very well. If this had been a loop, and the coefficient variable had been calculated for each iteration, a good compiler would have had no problems in optimizing that into a constant expression. However, the evaluation involves virtual functions and it could therefore be difficult for the compiler to do something about this, since the call is dynamically dispatched at runtime. Also, it should be noted that these are optimizations that in our model centers around multiplication and division. The operations that by far takes the most time are transcendental math functions like *log*, *pow, exp* and *sqrt,* so we should maybe not expect huge differences. We show the results of this small optimization in Section 3.4.4.

Listing 3.7: Evaluation of the *BifurGaussian* function, calculating constant values for each evaluation.

```
inline Double_t evaluateLocal(const Double_t x, const Double_t mu, const Double_t sigmaL,
     const Double_t sigmaR) const {
  Double_t arg = x - mu;
  Double_t coeff = 0.0;
  if (arg < coeff)
  {
    if (TMath::Abs(sigmaL)>1e-30)
    {
      coeff = -0.5/(sigmaL*sigmaL);
    }
    else if (TMath::Abs(sigmaR)>1e-30)
    {
      coeff = -0.5/(sigmaR*sigmaR);
    }
  }
  return TMath::Exp(coef*arg*arg);
}
```

3.4 Results

We present the runtime/speedup results obtained from the TBB and OpenMP versions we have developed, with and without our own evaluation approach, in addition to the other

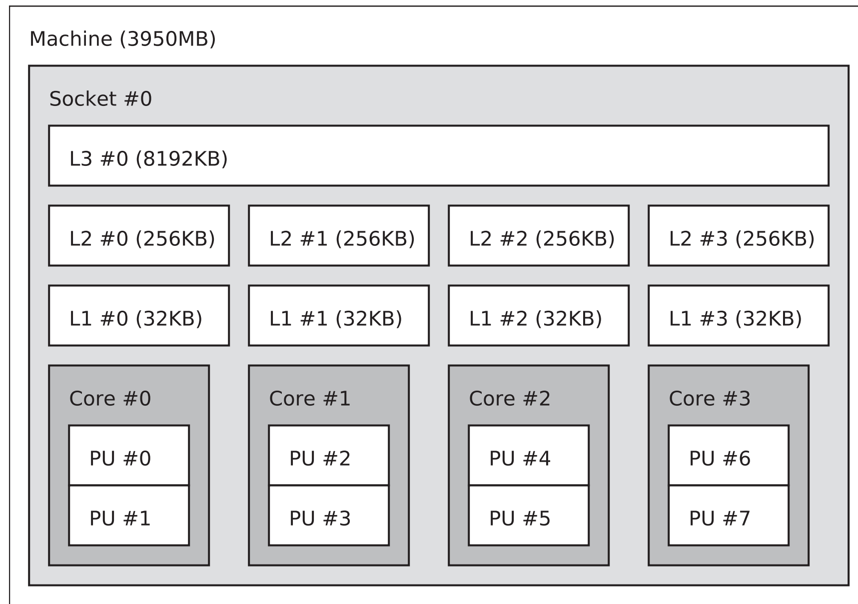


Figure 3.6: The topology of an Intel Core i7 965 CPU.

optimizations. We have concluded that OpenCL is not the way to go for us on the CPU side (atleast not yet), so we do not include results for that implementation here. Since we are aiming at commodity hardware primarily for now, we have benchmarked our different versions of MLFit on a modern machine featuring an Intel Core i7 965 CPU, running at 3.2 GHz and 2GB of DDR3 RAM. The Core i7 965 is turning into a commodity processor by the time this is written, but it is still reasonably high-end (although not Intel's flagship at the time being). This CPU has 4 physical cores, 32 KB of L1 cache, 256 KB of L2 cache and 8192 KB of shared L3 cache between the cores. The Intel Core i7 965 supports SMT (simultaneous multithreading), also called Hyper-Threading by Intel [36]. This means that it has the ability to physically execute 8 threads simultaneously and on a per-core basis (with 2 threads) try to latency-hide the memory accesses of each of the two, aiming to exploit the ALUs better. This can be thought of as hardware thread scheduling instead of plain OS/software thread scheduling. Figure 3.6 shows the topology of this CPU. The operating system used is Scientific Linux CERN 5 (SLC5), which is a modified version of Red Hat Enterprise Linux.

The benchmarks include runs for 10 000, 50 000, 100 000, 500 000 and 1 000 000 events. Each run is a pure *NLL* function evaluation done 100 times to achieve an accurate timing result, which means that the results in seconds we present (*NLL evaluation time*) is the time of 100 function evaluations. The timing facility used is the OpenMP function *omp_get_wtime*. Two calls to this function surrounds the loop that call the *NLL* function. Table 3.1 shows

$t_1(s)$	$t_2(s)$	$t_3(s)$	$t_4(s)$	$t_5(s)$	$\bar{t}$	σ	SE
4.0405	4.0341	4.0367	4.0260	4.0213	4.0317	0.0078	0.34%

Table 3.1: 5 timings for 100 eta'K model evaluations with 4 threads and 1 000 000 events.

the measurement of 5 runs with mean, standard deviation and the standard error (hereby abbreviated SE). The standard error is 0.34%, i.e. the timings are very accurate. We will use these numbers as a foundation for further timings throughout the thesis. In addition to targeting commodity hardware, we show how MLFit performs on a larger NUMA[2] machine.

3.4.1 TBB and OpenMP, explicit and implicit

It is interesting to see how TBB performs compared to OpenMP on a commodity machine (Core i7), with and without the new evaluation pattern we introduced, even if it has its downsides and is less suitable for MLFit than OpenMP. By this we mean that in the end MLFit must be implemented with OpenMP anyways, since it allows deterministic reductions. The new evaluation pattern has the potential to inflict on both performance in general as well as scalability, since the goal is to minimize the threading overhead by having one parallel entry point.

Figure 3.7 shows the runtime results for both versions with both the explicitly parallel and implicitly parallel evaluation pattern. The implicitly parallel versions perform significantly better in most cases, however, the most characteristic (and critical) observation is the huge performance degradation the OpenMP version suffers when N grows large. This is the manifestation of what we discussed in Section 3.3.2. TBB is implemented by threads taking tasks in blocked ranges, and these ranges are much smaller than the static partitions used in the OpenMP version. Therefore the TBB version (atleast the implicitly parallel) is more robust to larger amounts of data (and thereby higher cache loads) for these implementations. We can note that the general scalability of the application so far is very bad. We can also note that SMT only has an effect on small numbers of N for these versions.

[2]NUMA stands for *Non-uniform memory access* and is a term used for many-socket systems. In these systems memory access latencies varies with respect to which CPU accesses which memory bank. When for instance placing 4 SMP nodes in one machine, it is logical that node number 1 will spend more time loading memory from node 2's memory bank than from its own.

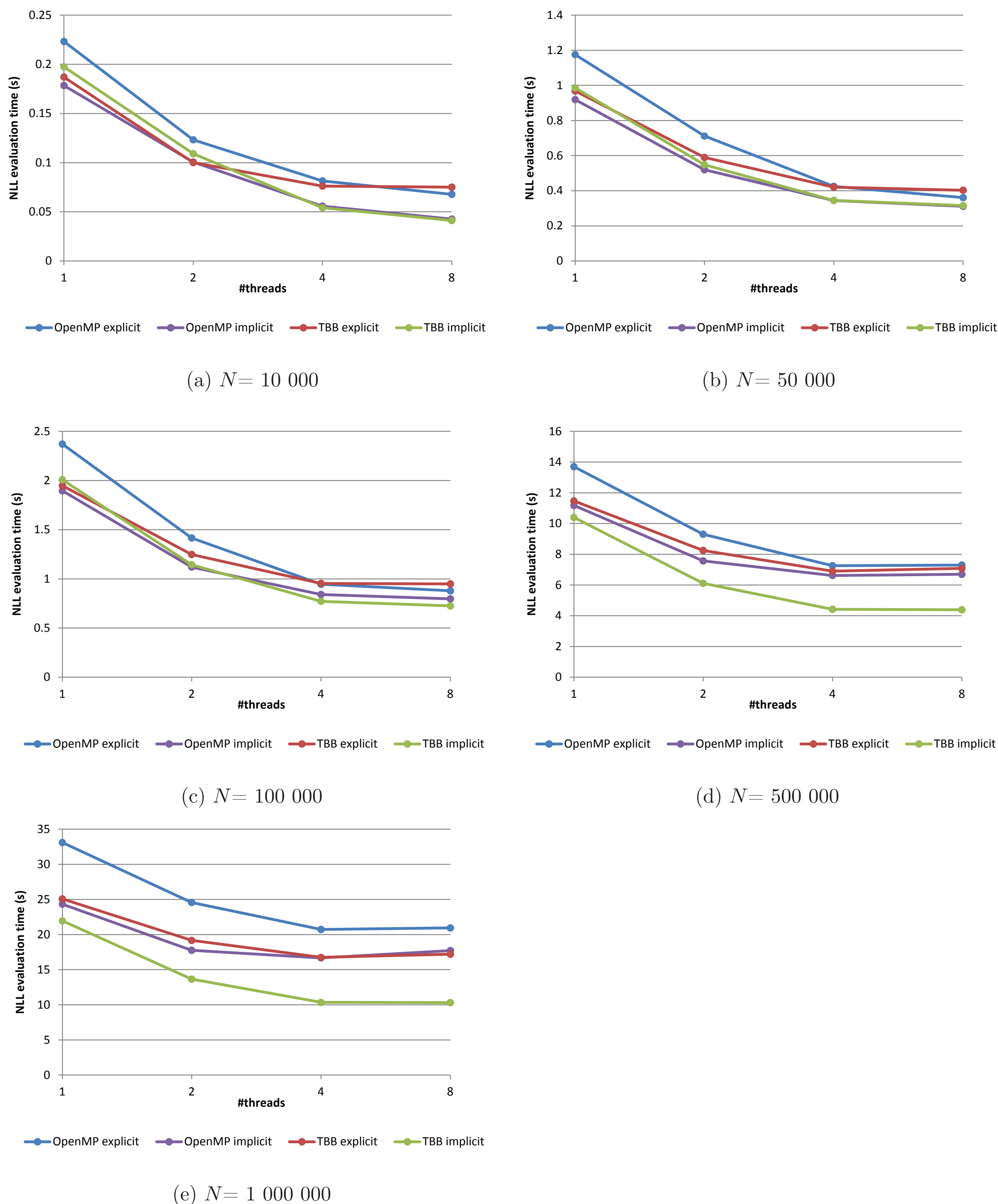
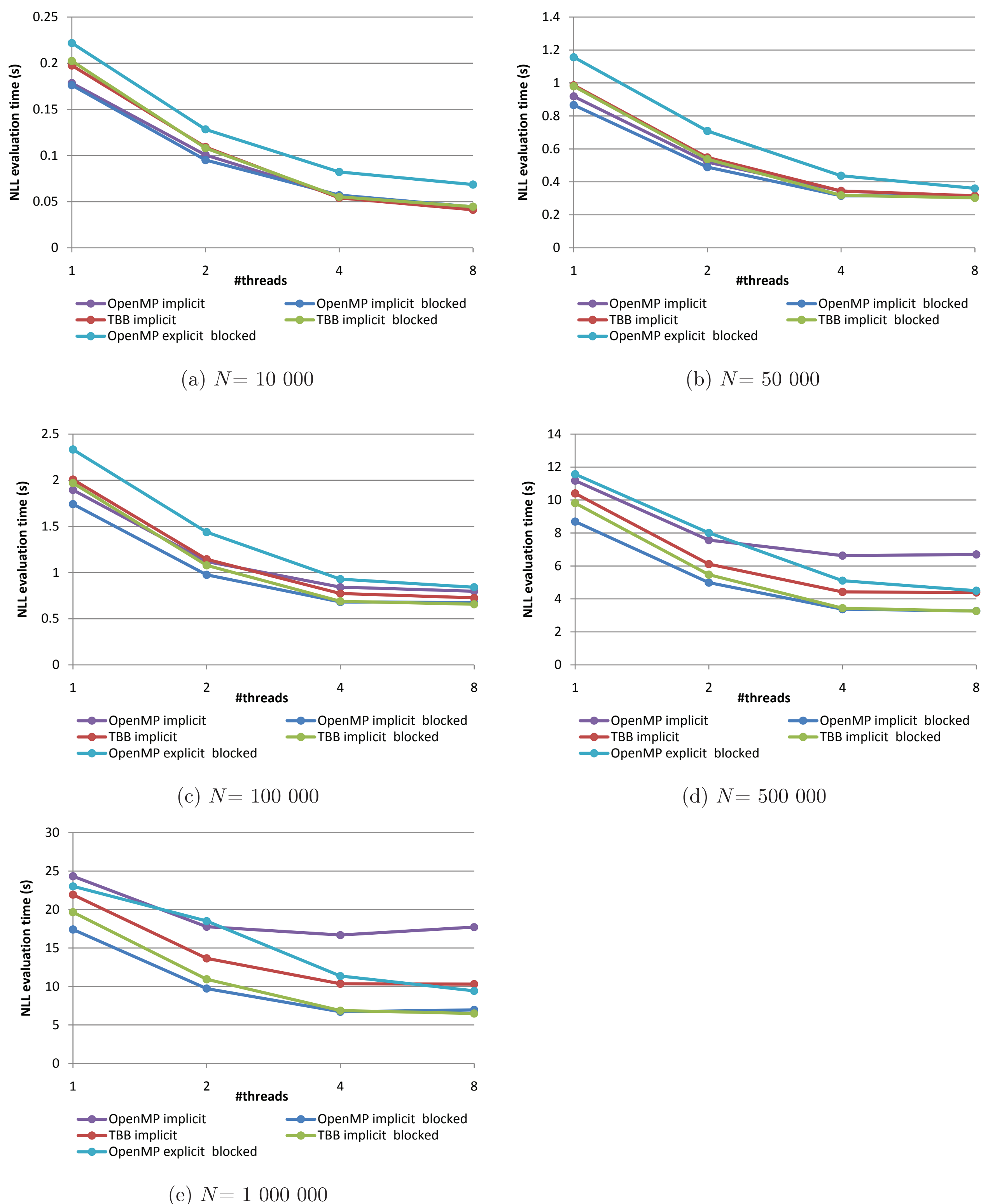


Figure 3.7: OpenMP version versus TBB version on the Intel Core i7 965. Both explicitly parallel and implicitly parallel.

3.4.2 Block splitting

To improve the bad performance reflected in Figure 3.7, a top-level block-splitting was introduced. This directly targets cache misses, since the application will operate on smaller chunks of memory at a time as explained in Section 3.3.2. The performance gain by doing blocking is of course highly dependent on the block sizes. We have experimented with a broad range of different values, but in the end (for this processor) one reaches a plateau where it does not matter much whether one increases or decreases it with e.g. 1000 elements. We found by experimentation that some of the best choices for the blocksize was 1000 for the OpenMP version and 5000 for the TBB version. However, this comparison is not about being 100% fair against any of the two implementations. It is more interesting to see if there are some surprising trends. Block size will of course be modifiable by the user of the application if he has any interest in experimenting with that himself. The size of the blocks depend on the hardware and the size of the model.

The results of this optimization are shown in Figure 3.8. For a low number of N, we do not see much difference between the versions (as expected), but when N grows large we clearly see the power of blocking. The blocked OpenMP implicit version is superior to the non-blocked, and is the fastest of the five in nearly every case (although in general with small margins with respect to TBB implicit, but this is also as expected). This was exactly the goal of this optimization.



(a) N= 10 000

(b) N= 50 000

(c) N= 100 000

(d) N= 500 000

(e) N= 1 000 000

Figure 3.8: Implicitly parallel versions of OpenMP and TBB on the Intel Core i7 965, with and without block splitting. The blocked explicitly parallel version is also included to compare with the blocked implicitly parallel one.

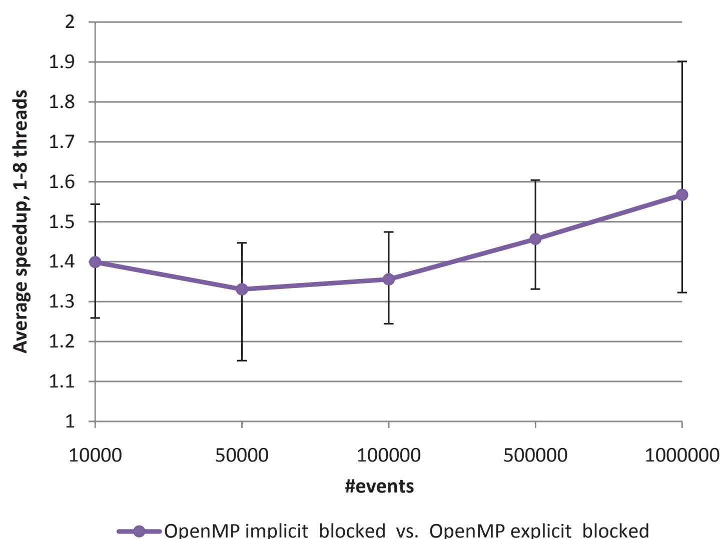


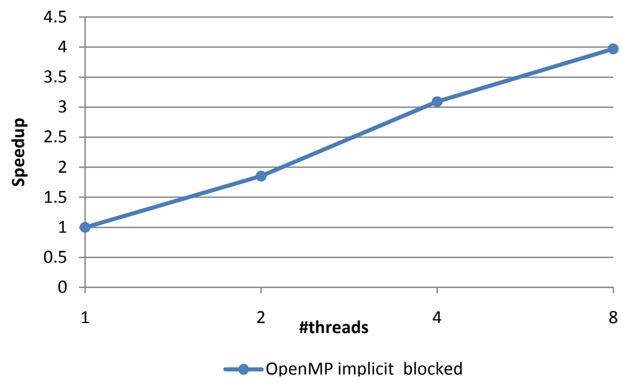
Figure 3.9: The average speedup of going from an explicitly parallel to an implicitly parallel evaluation, with errors. The average is taken over all threads, and is shown for different values of N.

How much was it in “total” possible to gain on using the new evaluation approach? Figure 3.9 gives the *average* speedup for the blocked implictly parallel vs. the blocked explicitly parallel version for all number of threads with respect to the number of events. The most important observation is that it seems to increase as N grows large and it is when N grows large that commodity systems need all the power they can get. We do not mean to analyze this speedup very thoroughly, the main point is that the new evaluation strategy is faster and therefore we will use it. Error bars are included to give some more information about the potential.

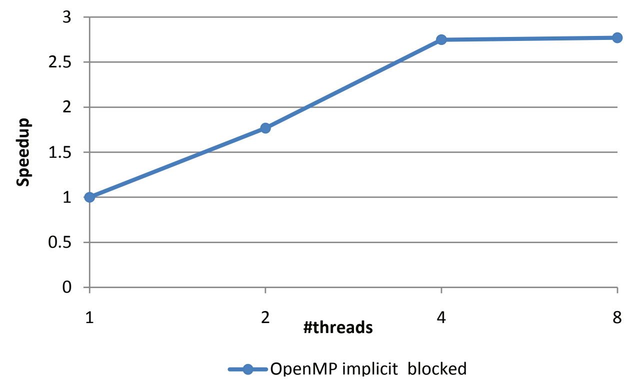
3.4.3 Scalability so far

We have converged on a version to use for further optimizations, which is the blocked implicitly parallel OpenMP version. Figure 3.10 shows a scalability plot for this version. The scalability has improved, but there is still much to improve before it will be good, since we only have about $\sim 2.5x$ speedup with 4 threads when N grows large. SMT obviously works well for 10 000 events, but after that the effect is almost negligible in every case, and leads to slightly loss of performance on 1 000 000 events. However, an important property of these plots is the clear pattern as N grows. The speedup is nearly constant with respect to N, which means that cache blocking works as assumed. Cache misses could be a problem still,

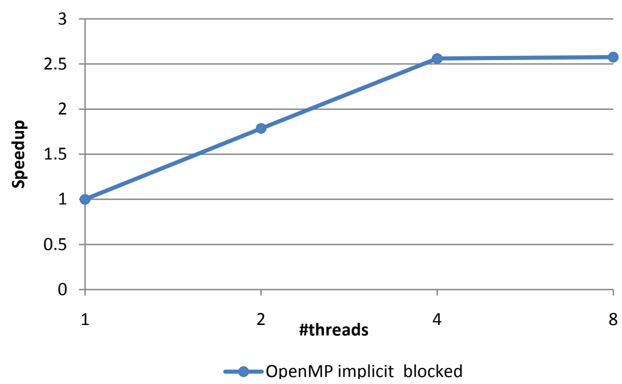
but at least it does not increase as the amount of data is increased. This is a quite good starting point to do further optimizations from. The most obvious difference between the plots is the case with 10 000 events compared to the rest; the speedup is substantially higher and this is because the data reside higher up in the memory hierarchy, e.g. fits to a higher degree in L3 cache instead of memory. Remember that the L3 cache is shared between the cores on this processor, as shown in Figure 3.6.



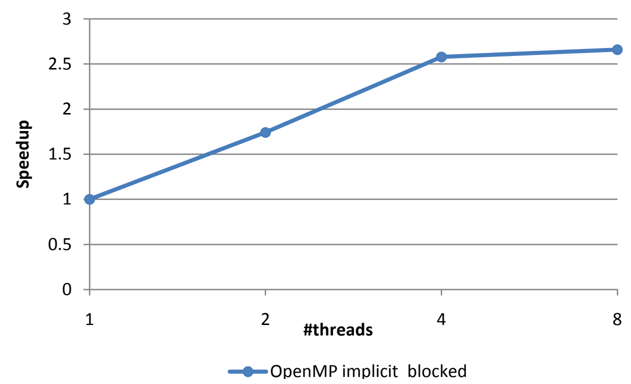
(a) N= 10 000



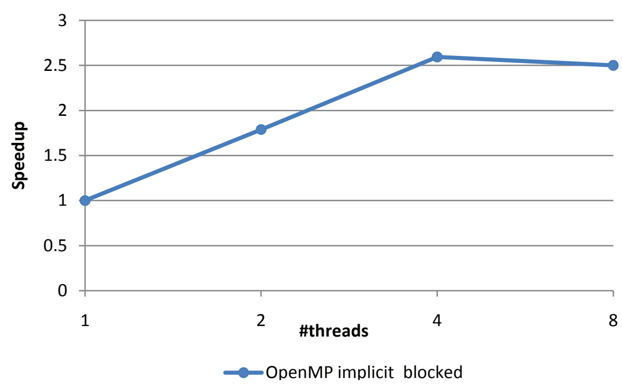
(b) N= 50 000



(c) N= 100 000



(d) N= 500 000



(e) N= 1 000 000

Figure 3.10: Scalability of the OpenMP blocked implicitly parallel version.

3.4.4 Result propagation, loop fusion and constant expressions

By doing these loop fusions and propagation of results, we do not doubt that it will go faster, but we are first and foremost interested to see if this has meant an increase in *scalability*. The results from this optimization was, as the last scalability results, more or less independent of N. Because of this we present a scalability plot in Figure 3.11. This shows the average scalability for all numbers of events for 1, 2, 4 and 8 threads. Note that we denote the edition utilizing both loop fusion and result propagation as *loop fused*, for convenience. Combining the block splitting with fusion of the composite loops and result propagation in addition to the final normalization loop for each PDF, the application is finally reasonably scalable. The plot crawls quite near the plot for ideal speedup, but it is still something that limits it somewhat. When using more than one thread, there exists a small OpenMP overhead, which we have seen using VTune, and we suspect that this is one of the major factors. It is also important to remember that the operating system occupies one of the cores slightly, which can become significant when using all cores. We have carefully measured the serial fraction of the program to be almost negligible, so the impact of Amdahl's law should in general be small. One has to remember that this is a very non-trivial application to optimize, and that it is memory-intensive; it is not directly clear when threads access memory since they "live their own life" when they evaluate the tree. The RooFit style of creating PDF models targets programmability mainly, and we have tried to "force" performance into it. We have applied optimizations that seemed logically reasonable to do, and it is very rewarding to see that the optimizations so far has lead to an if not perfectly scalable, then at least nearly perfectly scalable edition of MLFit on commodity machines.

Having the scalability characteristics in mind, we present the actual speedup of the final version in Figure 3.12, with the average values over 10 000, 50 000, 100 000, 500 000 and 1 000 000 events. We can see from this plot that the new evaluation pattern (implicit) improved the evaluation with everything from 25% to 40% overall, when considering the average with respect to the *number of threads*. Note that Figure 3.9 also tries to show this, but it shows it with respect to the *number of events*, with the average for all threads. Applying cache blocking increases the performance significantly. The problem is that with blocking only, the application will still not scale properly, and this is something that it is non-trivial to give a good answer to, since the operations in the composite PDFs should be blocked now. Therefore it must be something very sub-optimal with the memory access pattern that occurs. All these problems are to a large degree solved with the fusion optimizations and result propagations, and by pre-calculating constant expressions we can in the most profitable case (8 threads)

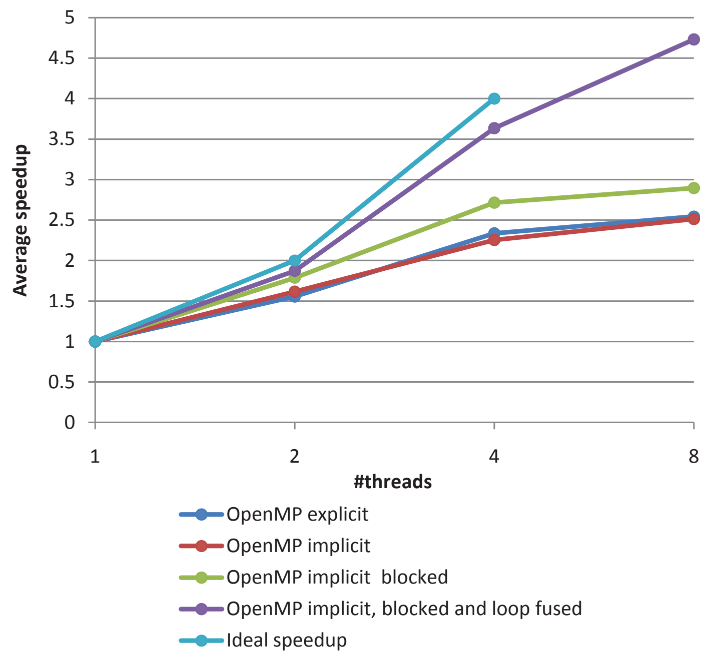


Figure 3.11: Scalability of the most important OpenMP versions.

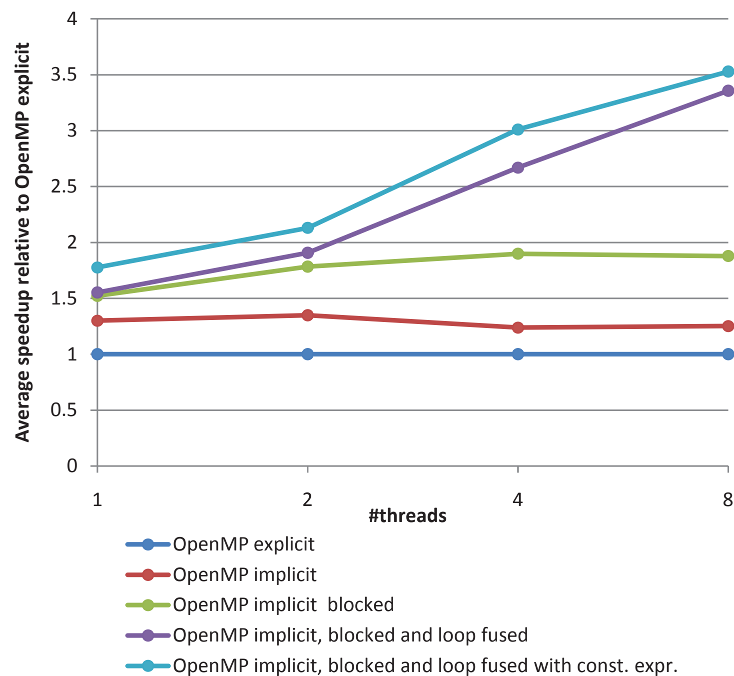


Figure 3.12: Speedup relative to the OpenMP explicit version.

achieve an average speedup of $\sim 3.5x$ compared to the OpenMP explicit version. Note that these are *average* measurements. We would like to add that the speedup of the final version compared to the OpenMP explicit version on 8 threads and 1 000 000 events is $\sim 6.45x$. Of course this is also lower than $3.5x$ in other configurations ($\sim 1.45x$ is the lowest we have measured). By pre-calculating constant expressions we gain anything from 5% to 14%, which is nice, considering the minimal effort to implement it. We note that we actually gain quite a lot on SMT with the loop-fused versions, but lesser with the final one than the one not using constant expressions. Obviously this means that the processor is able to do latency-hiding of memory accesses well when there are done a lot of computations. The constant expressions version is doing fewer computations while keeping the memory traffic as before, and therefore it is reasonable that SMT will be less significant. Anyways, SMT should just be considered as a pure “bonus”. Jarp et al. reports in [23] a single-core speedup of $4.5x$ for the explicitly parallel version of MLFit compared to the original RooFit. By Figure 3.12 our speedup is now around $1.75x$ compared to that version. An estimate of the final version compared to RooFit running on a single core will then be $\sim 4.5x * \sim 1.75x \approx 7.8x$. On top of that comes a scalability of $\sim 3.6x$ on 4 cores, and good utilization of SMT ($\sim 4.7x$).

3.4.5 NUMA results

We have in the sections up until now run MLFit on a single-socket machine, and we present in this section the results from running on a dual-socket NUMA machine with up to twice as many threads. When running on more than one socket, the concept of *affinity*, or *pinning* can be very central. By this we mean which thread is scheduled to run on which core. Executing the program naively will put all this responsibility on the operating system. Depending on the application, this can incur performance penalties since the effect of scheduling a thread on another socket can mean loosing all data loaded into the cache in the CPU on the former socket (cache misses). More specifically, if one thread allocates some memory area, this memory area will most probably be allocated in the memory bank corresponding to the socket of the CPU the thread is currently running on. However, if this thread now is scheduled onto another socket it might incur large penalties since the thread must go to the former CPU’s memory bank to fetch these data. This can therefore also have a large impact on reliability of timing results. Another important point is that it will be more cache efficient to evenly distribute threads across sockets, “waiting as long as possible” with filling a socket with threads. Intel OpenMP (and maybe other implementations) supports a flag/environmental variable named *KMP_AFFINITY*. This flag gives the user the opportunity to control thread

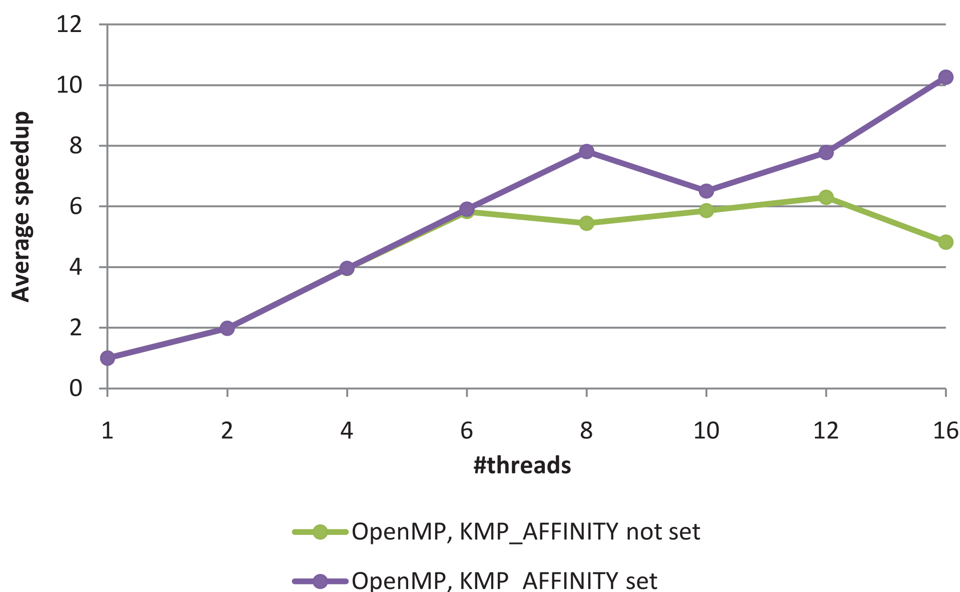


Figure 3.13: MLFit scalability on a dual-socket Intel Core i7 machine.

affinity, i.e. locking a thread to a specific core. Figure 3.13 shows the scalability running both with and without KMP_AFFINITY set. The speedup is as good as perfectly linear with the affinity flag set (this also proves that Amdahl's law is close to insignificant in our case). However, leaving scheduling to the operating system involves a large penalty when the machine starts to get filled. This demonstrates the importance of pinning. Note that this is on an 8-core machine and that we run SMT on top of these 8 threads. SMT gives around 25% speedup when pinning is done properly and the machine is filled.

3.4.6 Preliminary conclusion

We have landed on an implementation that suits our needs for future development. It can be assumed that the application will scale reasonably well on commodity processors, and also on larger NUMA machines, atleast when taking pinning into consideration. VTune profiles have shown that OpenMP overhead represents a few percent of the total running time for the whole application (this can be very variable), and it increases slightly when using SMT. This is important, since it means that it will be difficult to improve it significantly using other threading mechanisms unless they have close to zero overhead in every case. In general this chapter has been more about low-level optimizations than what kind of threading one applies.

Function	Time
__svml_exp2.N	47.6%
__svml_log2.L	6.0%
PdfPolynomial::evaluateAndPropagateInit	6.0%
PdfArgusBG::evaluateAndPropagateInit	4.8%
NLL::GetVal	4.6%
PdfGaussian::evaluateAndPropagateProd	3.9%
PdfBifurGaussian::evaluateAndPropagateProd	3.3%
PdfPolynomial::evaluateAndPropagateAdd	2.9%
PdfArgusBG::evaluateAndPropagateProd	2.6%
PdfGaussian::evaluateAndPropagateInit	2.0%

Figure 3.14: A VTune hotspot profile of the final version, running 1 000 000 events on 4 threads on the Intel Core i7 965.

It may be possible to do further optimizations with regards to memory access, threading overhead, and in general trying to achieve close to perfect scalability on single-socket machines with a CPU with limited cache size. However, we will say stop for now because there are other areas in our work that is more important than fine-tuning. The most important optimizations are the fundamental ones like implicitly parallel evaluation, cache blocking, loop fusions and result propagation. We mean that it is not possible to add any more fundamental/architectural optimizations without breaking the interface of RooFit. Figure 3.14 shows a VTune hotspot profile for the final version. We can see that almost half of the time is spent doing exponentials, which is a really good sign. The functions further down the list are the evaluation functions, and with the optimization of constant expressions we can be sure that they do only what they *must* do and nothing more.

Chapter 4

MLFit on GPUs

Since modern computers often are equipped with programmable graphics processing units, it will be advantageous if MLFit can benefit also from them. In this chapter we present the implementation of MLFit for GPUs using OpenCL. MLFit was actually implemented in CUDA before this work was started, but we will here focus on OpenCL and on the eventual optimizations that is possible to implement. The key reason for using OpenCL is the portability between vendors. It is not ideal to be bound to NVIDIA GPUs just because the software technology dictates that. Another important point is that MLFit for CPUs now has a completely different performance profile, so this chapter will also include a thorough performance comparison.

4.1 Graphics processing units

In recent years, it has become more common to use graphics processing units (GPUs) as general co-processors. GPUs are designed to process image textures, where each pixel is independent of all others, but identical operations are used to process each. Therefore, general purpose algorithms that can be classified as SIMD perform efficiently on GPUs. On GPUs, a larger fraction of the silicon is spent on ALUs instead of caches and control-flow mechanisms [31]. GPUs can in some cases be much faster than price-equivalent CPUs, but not necessarily, since the GPU is an accelerator connected through a bus (the PCI-Express bus for instance). Transferring data back and forth on this bus can lead to high communication overhead, which again could make a GPU performance-wise obsolete compared to a CPU in some scenarios. The GPU is a massively parallel throughput device, that aims to deliver massive memory bandwith and parallelism by sacrificing low latency. Just as a comparison, the Intel Core i7

965 CPU has a theoretical maximum memory bandwidth of 21 GB/s [37] while an NVIDIA GTX470, which is almost equally priced by the time this is written, has a memory bandwith of 133.9 GB/s [38]. One of the prime drivers for the popularity of GPUs is the video game market. Modern video games are highly demanding when it comes to rendering capabilities, and since computer games are very popular, the market is enormous. Graphics operations done on pixel buffers are nothing more than plain arithmetic operations, so there is no fundamental difference between rendering graphics to the screen or performing some kind of scientifical computation on those data. A GPU is a many-core processor with up to thousands of cores, usually running on a lower frequency than price-equivalent CPUs. This has to do with power consumption/temperature, and is in general a *scale out* approach as opposed to a *scale up* approach. These cores are able to execute thousands of threads with effective hardware context-switching, and the instructions run should stem from SIMD-like code for best performance, avoiding branching for instance. Each core executes an instruction in a lock-step fashion, and eventual branch divergence will be computed sequentially, which again will lead to a serious peformance decrease. NVIDIA CUDA uses a stack-based reconvergence algorithm for this [39], but this will of course depend on the implementation when using OpenCL.

Another major difference between CPUs and GPUs is the caches and memory. While the CPU often has a hierarchy of data and instruction caches (differing mainly by size, access times and if they are pure instruction/data caches or combined ones), GPUs have a number of different types of memory. We list (based on previous work presented in [40]) the different memory types on an NVIDIA GeForce GTX470, which is one of the two GPUs we will do benchmarks on in this chapter. The different memory types on the GTX470 are:

- Global memory
- Local memory
- Constant memory
- Texture memory
- Shared memory
- Registers

The most obvious type of memory is **global memory**, which is a large and relatively slow memory that each thread in every block can access. As an example, NVIDIA states that

the latency associated with a read from CUDA global memory is everywhere from 400 to 600 clock cycles [31]. This memory is off-chip. **Local memory** is the same kind of memory as global memory, and are often used as a register spill space if the number of registers required by the threads exceeds the number of registers available. Note that this memory is per-thread. **Constant memory** is a small cached read-only memory. It is therefore faster than global memory, but unwriteable unless one copies data from host memory to it. Since it is small, it can only be used for a limited amount of data, typically constant values that are to be reused globally. **Texture memory** is memory with a cache optimized for 2D access. It is therefore appropriate to use for reading textures/images, but is also read-only. **Shared memory** is shared among threads in a thread block, and is on-chip, thus very fast. Shared memory is divided into blocks called memory banks, where each bank can be accessed simultaneously. NVIDIA states that shared memory is as fast as accessing a registry as long as no bank conflicts occur (i.e. two threads access the same bank at the same time) [31]. Finally, **registers** are the fastest form of memory on an NVIDIA GPU, but they are limited in numbers so they should be used wisely. The compiler will try to use registers for frequently accessed variables.

4.2 Implementation

Since MLFit already was implemented using CUDA, the GPU part of our work is significantly smaller than the CPU part. The description of the work done using CUDA is thoroughly described in [23], but with the major changes done to the OpenMP version of MLFit comes also entirely different results. There is one important point to mention though. The new OpenMP version of MLFit (implicitly parallel) is based on a directly top-down evaluation of the tree. However, this is not possible using the GPU, since it really does not fit with the way a GPU does computations. OpenCL is plain C, and the concept of virtual functions for instance is something unknown. We will therefore have an explicitly parallel evaluation of the tree for the GPU. An interesting fact is that the OpenCL standard is designed to support asynchronous execution. This is reflected in the way API calls are named, i.e. *clEnqueue-*<call>. A good example is the *clEnqueueNDRangeKernel* call for enqueuing a kernel for execution. This means that if the underlying OpenCL implementation supports this, the evaluation of the tree will be non-blocking, i.e. kernels will just be enqueued for execution on the GPU. Then an implicit synchronization must occur at the end when the final reduction result is to be copied over the bus. Since the OpenCL version of MLFit is explicitly parallel,

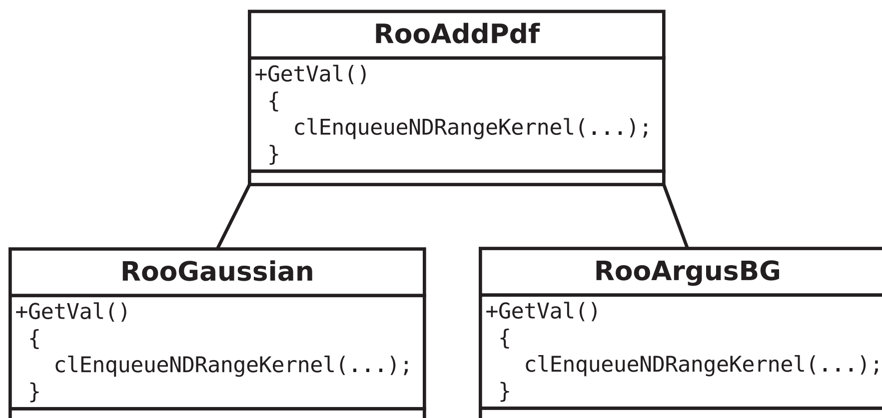


Figure 4.1: An explicitly parallel evaluation with OpenCL in MLFit.

each PDF has the ability to execute a kernel that corresponds to the given computation. This is illustrated in Figure 4.1.

4.3 An initial experiment

Before running MLFit on a GPU, it is interesting to explore the characteristics of simple computations, as well as computations involving transcendentals, on a much smaller example. We use two GPUs through this chapter; one NVIDIA GeForce GTX470 and one AMD Radeon HD5870. The theoretical peak performance properties on these cards are from a hardware perspective far from equal, as showed in Table 4.1. Most numbers in this table are taken from [38] and [41], which are the respective product web sites for these two GPUs. NVIDIA do not list the peak performance of the GTX470, but we can give an estimate of the peak performance of the NVIDIA Fermi GT100 chip, which is the actual chip residing on the GTX470 card. To be precise, we hereby denote a floating-point operation as *FLOP* and many floating-point operations as *FLOPs*. Floating-point operations per second is denoted $\frac{FLOPs}{s}$, *FLOPs/s* or *FLOPs per second.* The Fermi GT100 chip consists of Streaming Multiprocessors (SMs). If we denote the ALU clock speed by f and the number of SMs with m, the theoretical maximum single-precision performance could be expressed as $f * m * (32 * 2)$. The constant numbers stem from the fact that the NVIDIA GTX470 has 32 ALUs per SM and that each ALU has the potential of doing an FMA[1]. Supplying its clock frequency of 1215 MHz and its SM count of 14, its maximum theoretical GFLOPs per second becomes approximately 1088. This is a purely theoretical number and should be interpreted as just that. The

[1]Fused Multiply-Add. The ability to do one multiplication and one addition in one clock cycle

	NVIDIA GeForce GTX470	AMD Radeon HD5870
Number of ALUs/scalar cores	448	1600
Core clock	1215 MHz	850 MHz
Peak single-precision performance	1088 TFLOP/s	2.72 TFLOP/s
Peak double-precision performance	$1088 * \frac{1}{8} = 136$ GFLOP/s	544 GFLOP/s
Memory bandwidth	133.9 GB/s	153.6 GB/s
Memory size	1.28 GB	1 GB

Table 4.1: A specification comparison between the NVIDIA GeForce GTX470 and the AMD Radeon HD5870.

theoretical peak double-precision performance for the GT100 chip is half the single-precision performance, or 544 GFLOPs per second. However, NVIDIA state that they have restricted the double precision performance to 1/8th of the single-precision performance on consumer-level GeForce cards, to get buyers that need high performance and double-precision to buy their Tesla series instead. The Tesla series are not directed at gamers, but towards professional HPC clients, and are many times more expensive. This should mean that the HD5870 theoretically could perform 4 times as good as the GTX470 in double-precision calculations. The major reason for us to use simple bench examples is that we believe this will simplify reasoning. We are interested to see how the cards compare, and if computations involving transcendentals are memory-bound or compute-bound. The main motivation for this is to find out what kind of hardware would be most suitable for users of MLFit (but also GPU applications in general). If AMD cards can perform 4 times as fast for approximately the same price, the choice of vendor should be obvious.

For timing kernels we use OpenCL kernel event timing, which can be used by passing appropriate flags for the command queue used to execute the kernels and by submitting a *cl_event* struct in the kernel call. The timings can after execution be obtained by OpenCL API calls using this event struct. The timings have been very accurate by our experiences, but the accuracy is of course vendor/implementation dependent. The workgroup size for the kernels in the following tests is set to 64 if the kernels involve transcendental functions, and 128 if not. More on these sizes in Section 4.4.

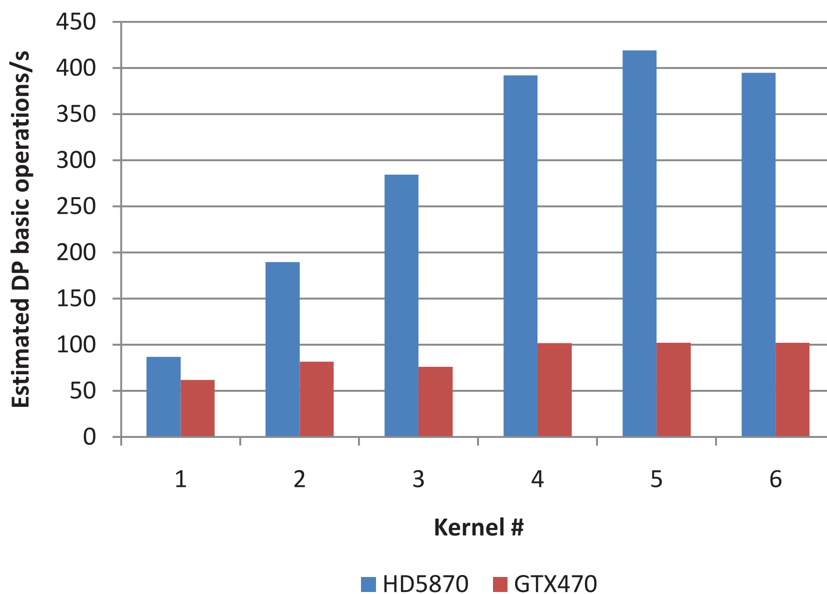
Additionally, we want to introduce our own measure, since we do not know how a "FLOP" is defined by either AMD or NVIDIA. We introduce the term *basic operation*, which can be either an addition or a multiplication. Of course, we assume that a basic operation is *comparable* to a FLOP on both cards. It is mainly the eventual *difference* in performance between the cards we are interesting in looking at.

Kernel #	1	2	3	4	5	6
Appr. DP basic operations pr. iteration	10	22	91	2101	21001	210001
Appr. Total DP basic operations	$1.0 * 10^7$	$2.2 * 10^8$	$9.1 * 10^8$	$2.1 * 10^{10}$	$2.1 * 10^{11}$	$2.1 * 10^{12}$

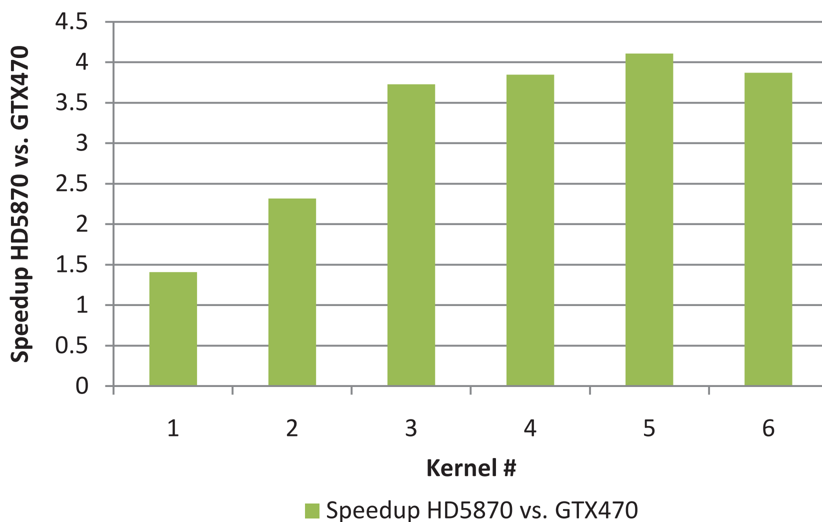
Table 4.2: DP basic operations counts for the kernels listed in Appendix A.

4.3.1 Measuring the double-precision basic operations

The theoretical limits of these cards are known, but it remains to see if this performance can be utilized. Appendix A shows a program listing of some basic OpenCL kernels. We want to see the effect of increasing arithmetic intensity when comparing the two cards, using double-precision (DP) floating point numbers. We count basic operations, and we ignore the overhead associated with e.g. loops. We do not claim that these measurements are completely accurate, but we think they are good approximations. These kernels are run with 10 000 000 elements, to be sure to saturate the GPU bandwidth. We have carefully tested that increasing this number has a negligible effect. Table 4.2 shows the approximate DP basic operation count for each kernel, and the approximate total number of DP basic operations when running with 10 000 000 elements. Now, based on this table, we can measure the runtime of each kernel and find an approximate value for the total DP basic operations per second each card performs. The results are shown in Figure 4.2. Apparently, the theoretical numbers seem to be close to true also in practice, atleast when the arithmetic intensity grows. The speedup for the HD5870 converges around $4x$ which is on par with the specifications for the two cards. The GTX470 peaks at around 100 DP basic operations per second , or approximately 81% of peak performance if we let basic operations and FLOPs be approximately equal. The HD5870 peaks at around 420 basic operations per second, or approximately 84% of peak performance. Clearly, global memory is a *huge* bottleneck for kernels with low arithmetic intensity.



(a) Performance.



(b) Speedup.

Figure 4.2: The benchmark results for both NVIDIA GTX470 and AMD Radeon HD5870 running the kernels in Appendix A.

4.3.2 Performance of computations involving transcendentals

In this section we set up some kernels involving transcendental math functions. We then keep the number of memory reads/writes constant while increasing the arithmetic intensity

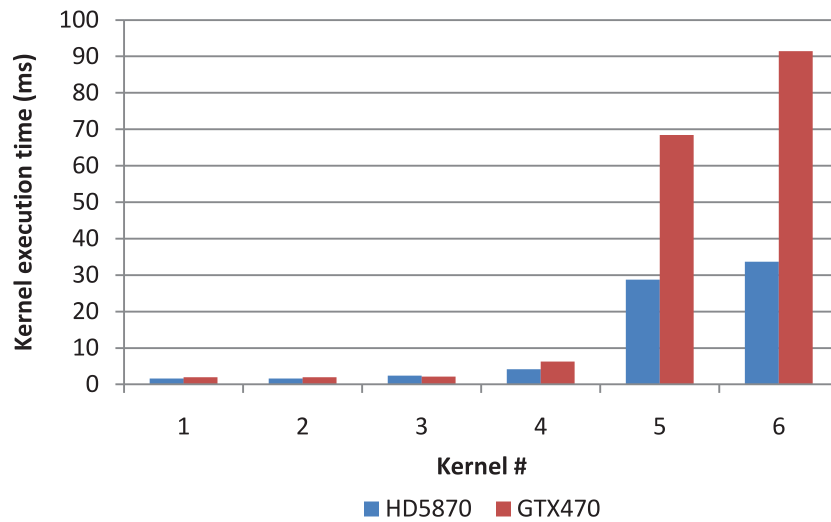
of the kernel, as in the last benchmark. The test consists of the computational kernels in Appendix B. The idea is that these kernels are more similar to MLFit when it comes to arithmetic intensity than the kernels in Appendix A. *Kernel1* is just doing a simple addition. Then we progress to multiplication and addition before involving the exponential in *Kernel3*. Finally, we just increase the complexity arbitrarily while keeping the number of global memory accesses, which is 3. Some decimal factors are included in the expressions just to escape eventual common subexpression eliminations[2] done by any one of the two OpenCL compilers (AMD and NVIDIA). The results in Figure 4.3 show as expected quite obvious signs of the global memory bottleneck. The first three kernels are so computationally light that there is just a slight difference in performance between the cards (max 22%). This means that the computation is entirely memory-bound on both cards. The HD5870 might be faster in these scenarios because of lower latencies, but it is in general difficult to say much about that since we do not know for sure what the compilers do. When increasing the computational complexity, however, we can see a significant difference with the HD5870 peaking at a speedup of $2.71x$ in the best case. This means that the AMD card clearly is not utilized as good as in the FLOP benchmark, but this is reasonable taking the lower arithmetic intensity into account (the FLOP benchmark kernels have large loops and fewer memory accesses).

In general, these results suggest that it will be more beneficial to use AMD GPUs if a fit includes kernels with *very* high arithmetic intensity. The HD5870 is faster than, or approximately as fast as the GTX470 in every run. The kernels that the PDFs of the eta'K model consist of often include the exponential function or other transcendentals, but are not nearly as computationally expensive as *kernel5* and *kernel6*. However, this could be the case for future PDFs, and it is important to emphasize that. We refer to Appendix E for an implementation listing of all relevant kernels in the model we use.

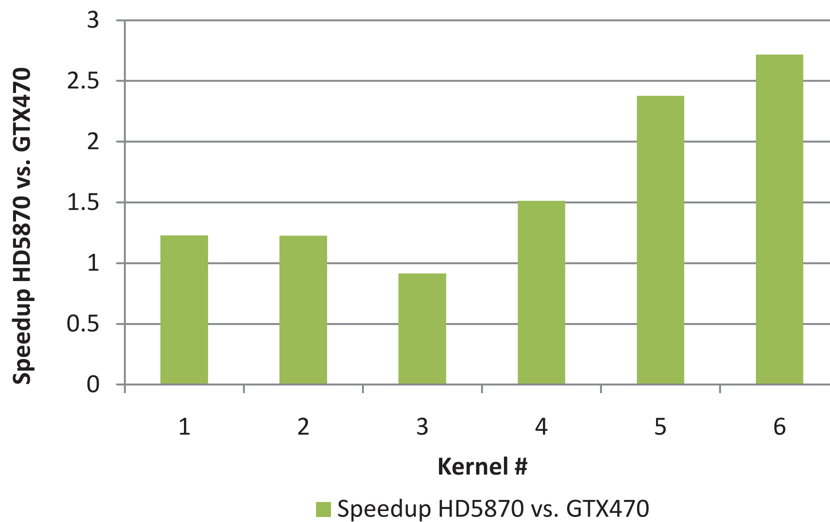
4.3.3 The effect of OpenCL vector types

We have already been through the usage of OpenCL vector types (Section 3.2.2) and we argued that it would not be an elegant solution for CPUs, based on the fact that using vectorizing compilers is more effective and that vectorized transcendental routines are not utilized by atleast the AMD APP SDK OpenCL compiler. Vector types can be significant

[2]Common subexpression elimination is a technique used by compilers to avoid computing expressions that already has been computed. If the expression *log(<identifier>)* occurs twice in a computation, it might well be that either one of the two compilers or both stores the result of the first occurence, and uses that in all the following occurences of it. This would be a potential critical source of timing errors in our case.



(a) Performance.



(b) Speedup.

Figure 4.3: Benchmark results from the kernels in Appendix B running on the NVIDIA GTX470 and the AMD Radeon HD5870 respectively.

for GPUs also, depending on the underlying architecture. AMDs architecture is different from NVIDIAs. The HD5870 consists of streaming processor units, each containing 5 stream cores based on a VLIW (Very Long Instruction Word) architecture, and is in many ways more similar to the vector registers on a CPU. These 5 stream cores can be divided into 4 general purpose ALUs and one special functional unit which has the ability to execute one single-precision transcendental calculation in one cycle. The SMs on NVIDIA Fermi cards, as we mentioned in Section 4.3 also have 4 special functional units (SFU), each able to execute a transcendental function on one single-precision number in one cycle, per thread [42]. In essence this means that the granularity of scheduling is different between the two architectures. AMDs approach relies on that the compiler does a good job utilizing the stream cores in the VLIW setup, while in the Fermi case, the compute capability reflected in the CUDA cores are more visible to the scheduler at run-time. In other words, while the GTX470 is based solely on thread-level parallelism (TLP), the HD5870 is based on instruction-level parallelism (ILP) in addition. Therefore, by our understanding, vector types will essentially only have a positive effect on the HD5870, atleast when memory accesses are aligned and memory coalescing[3] is achieved for the GTX470.

Figure 4.4 shows *kernel6* from appendix A run with and without using double-precision vector types for both the GTX470 and the HD5870. The results are as expected, except a slightly performance increase for the HD5870 when using vector types. It can be that the compiler became able to generate more efficient code, or that memory coalescing was exploited better because we use an aligned type. However, we knew that it was not possible to reach much higher since the theoretical peak already was reached (relative to the GTX470). Therefore, vector types on these GPUs when using double-precision accuracy hardly makes any sense.

Figure 4.5 however, which is the same case only using single-precision (SP) instead, shows completely different results. Using ordinary data types, the HD5870 is just performing a tiny fraction of its true capability (most probably only one of the ALUs). But when using the *float4* vector type provided by OpenCL performance jumps dramatically. This means that when running with appropriate kernels (compute-bound), and with single-precision accuracy, OpenCL vector types must be used (atleast in our case) to really exploit the computing power on the HD5870. This is disappointing, since there should be no problem for AMD

[3]Memory coalescing is the concept of that memory accesses from more than one thread is combined into one memory access. This is fulfilled for wavefronts if some requirements are fulfilled (in particular aligned memory access), depending on the compute capability of NVIDIA GPUs [31]. It is also relevant for AMD GPUs.

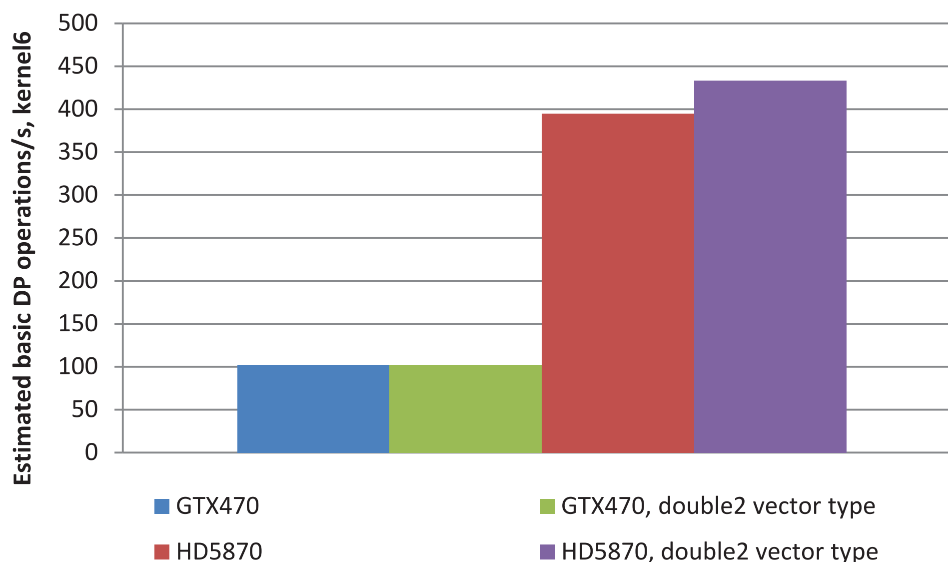


Figure 4.4: Kernel 6 from appendix A run with and without using double-precision vector types for both the GTX470 and the HD5870.

to implement this automatically. They know that kernels in most cases follow the SIMD structure, and in essence this means that to reach absolute portability, vector types must be used for both cards, avoiding different kernels. We believe the small performance increase for the GTX470 can be because of increased memory coalescing, just as for the HD5870 in the previous example.

The primary conclusion of this test is that as long as we use double-precision, there is not very much to win on using vector types when running with AMD cards. And in our case we are completely memory-bound, which means that the stream cores are starved anyways. In other words, vector types makes sense for kernels with very high arithmetic intensity doing basic arithmetic operations in single-precision accuracy. If transcendentals are used, these are left to the SFUs, and it does not make any sense to use vector types for them, taking the architecture into account. Therefore, in our case, vector types are not a topic for optimization, and is not included in the next section, which discusses GPU optimization possibilities for MLFit.

4.4 Optimization possibilities

There are a lot of different characteristics of GPUs, but MLFit is a computation-wise straightforward application. The general workflow is to load an element from a data array, calculate

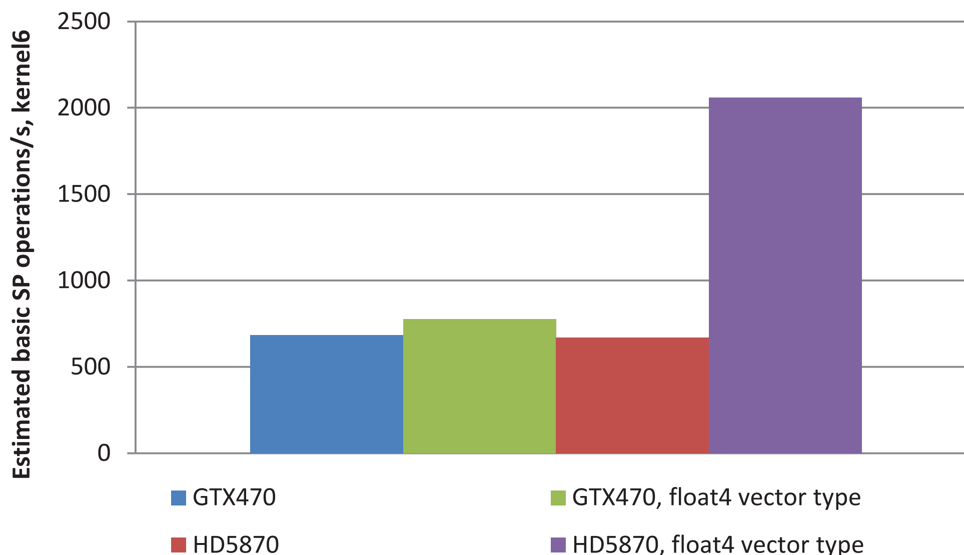


Figure 4.5: Kernel 6 from appendix A run with and without using single-precision vector types for both the GTX470 and the HD5870.

some function with this element as input, and put the result in a result array. In this section we describe the opportunities the GPU gives us to eventually optimize these calculations.

4.4.1 Single-precision

Since GPUs originally was made to operate on and render textures, they are optimized for single-precision since that is sufficient for such computations. We have already seen the huge performance differences between single-precision and double-precision on these cards in Table 4.1. However, in scientific applications it can be necessary to use double-precision to achieve a sufficient accuracy. This project is not only about performance, since it is considers software in production. Tests we have conducted show that MINUIT will not converge when we switch to single-precision accuracy for the CPU version of MLFit (MINUIT uses double-precision accuracy exclusively). We have not looked any more into moving to single-precision accuracy, since it would involve exploring MINUIT extensively. And MINUIT is just one minimizer.

4.4.2 Parallel reduction

MLFit for CUDA was implemented with the final *NLL* reduction of the logarithm values done with the CPU. This was to ensure deterministic reductions. However, there exist algorithms for deterministic parallel reductions on the GPU. These algorithms are often based on a

workgroup-wise leveled reduction, using a kind of recursive halving of the elements containing partial sums, often giving good algorithmic complexities. They also typically exploit shared memory to improve memory access times. This is ideal since we want to utilize as much of the GPU as possible. The main goal of moving reduction onto the GPU however, is to get rid of the transfer of variable amounts of data from the GPU to main memory (over the PCI-Express bus). With a reduction done with the CPU and 1 million double-precision events ($\sim$ 8 MB), the transfer time can be substantial compared to the time spent on evaluation. This is a serious bottleneck and is really not beneficial. By doing the reduction on the GPU and sending 1 final value, the transfer time is much lower, but includes some overhead still (PCI-Express latency). We have therefore used one of the parallel reduction algorithms provided by code samples from NVIDIA.

4.4.3 Texture cache

Modern GPUs have (as mentioned in Section 4.1) a special kind of memory optimized for read-only 2D access and used to access texture data in an efficient way. The source of the data that is computed is in our case a good candidate to be placed in this kind of memory, since it is read-only. We have implemented support for texture cache in MLFit, but tests we have conducted have shown that rounding problems occur when storing/retrieving data from it. In addition, we did not see any performance improvement. Note that texture-cache supports single-precision only. However, we have done some modifications to store a double-precision number as two single-precision numbers. This is a kind of unorthodox way of doing it, and it might be the reason for these rounding errors, as well as the lack of speedup. This could of course also be implementation specific, but we cannot take this risk since the results really have to be correct independent of which OpenCL implementation the user runs. We will therefore not concentrate any more on texture cache usage.

4.4.4 Result propagation and loop fusion

The technique of propagating results from children and up to the parents in the tree made significant performance improvements for the CPU version. This first and foremost had to do with lowering the memory traffic. It is therefore possible that this optimization would contribute significantly in a GPU scenario, since it is, judging by our initial experiment, almost guaranteed that we will be memory-bound in many kernel executions. Introducing result propagation in the CPU version implied a small penalty implementation-wise, in that

the code gets a bit more expressive in the evaluation functions of each PDF. For OpenCL however, the situation gets worse. In order to do result propagation calculation, we would have to have one OpenCL kernel for each composite operation, for each child PDF. At the moment this would mean two OpenCL kernels for each PDF (one for propagating additions, and one for products), and this is really destroying programmability. This project is not only about performance, but also maintaining programmability (as we have stated before), and we have decided that implementing this optimization for OpenCL is not a good way to go. We will therefore discard it, and keep the implementation as is. One might argue that this is unfair when comparing a CPU against a GPU, but this is not what this project is about. We want to have as much performance as possible without breaking programmability to a significant degree, and which specific device is fastest is of very little interest to us.

Fusing the normalization loop, however, is possible without any major modifications. We have implemented this by precomputing the reciprocal of the integral once on the CPU, and then sending this as an argument to every PDF evaluation kernel.

4.4.5 Constant expressions

As mentioned in Section 3.3.4, it can be beneficial to evaluate constant expressions when it is tricky for the compiler to do it. This is an even more central area when moving to the GPU, since the parallelism is implicit in the OpenCL programming model and there is more than one compiler involved. An OpenCL compiler will compile the kernels and it will of course be limited to the parameters that the kernel accepts. If we send pre-computed values as arguments to the kernel, the GPU would not have to do the calculation of these expressions for each of its threads. This will mean lower load on the available resources of the multiprocessors that schedule these threads. We have implemented this for the GPU version.

4.4.6 Occupancy

Our work with MLFit is a very general effort of optimization, at least for GPUs, where the architectures differs in a higher degree than CPUs from different vendors. It is unclear to us what kind of PDFs physicists will add in their analyses, and we do not know which GPUs they will use. NVIDIA defines computational occupancy as the *number of active wavefronts per multiprocessor to the maximum number of active wavefronts* [31]. This is a direct consequence of their underlying architecture, since this will be a trade-off between resource usage and scheduling overhead. If computationally expensive kernels use the majority of resources on

an SM, it is ideal to have small workgroups, increasing the probability of exploiting the SM, and leaving a bit more work for the scheduler. However, if fewer resources are used, larger workgroups can be scheduled since the probability of exploiting the SM anyways would be "high", of course depending on the numbers. This of course goes also for AMD, since both architectures implement multiprocessors containing scalar cores.

Because of the generality, and the potential amounts of different hardware MLFit will be run on, we have not concentrated deeply on tuning the workgroup sizes. However, we have used a kind of heuristic for determining it. The rule is that if a kernel contains a transcendental function, the workgroup size is set to a "low" number, in our case 64. If the kernel does not contain transcendentals, but rather only basic arithmetics, the workgroup size is set to a "slightly higher" number, in our case 128. These numbers has provided good results for both the GTX470 and the HD5870. The occupancy numbers for the GTX470 have ranged from 0.33 to 0.67 depending on the kernel. The speedup results are shown in the next section.

4.5 Results

We have run the same benchmark as we used at the end of Chapter 3, and we use the final optimized version for the CPU running with 8 SMT threads to fully utilize the Intel Core i7 965. By the time this is written, there seems to be a trend within the HPC community to publish GPU results compared to a non-optimized CPU version running on one core. The small survey in [43] , which in fact is about GPU usage in HEP, is a good example of this. By doing that, many applications will reveal artificial "speedups", giving a wrong impression of the actual computing capabilities inside a modern CPU. In a real scenario, this is really not interesting from any perspective. In such a comparison it is important to pay attention to both implementations, and really try to make each of them performant so that the hardware potential is utilized. Also, it is important to emphasize which price segment one is operating in. In our case, both the Intel Core i7 965, the NVIDIA GTX470 and the AMD Radeon HD5870 are comparable with respect to price, and could be regarded as commodity hardware (yet again, by the time this is written). Of course, we have to remember that our CPU and GPU implementation differ, because of the non-performance related requirements we have to take into account.

The results are shown in Figure 4.6. First of all, it is important to note that running with the GPU instead of the CPU is not beneficial for a low number of events. This is a direct

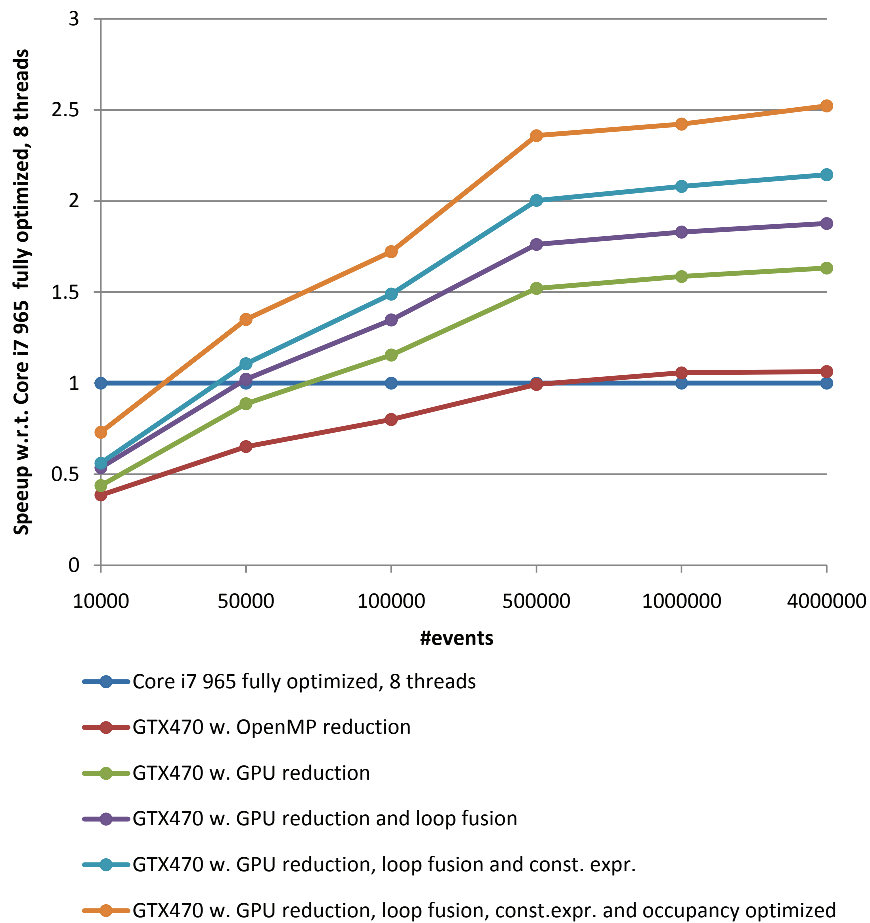


Figure 4.6: MLFit benchmark, comparison between CPU and GPU.

consequence of the need to copy the final value over the PCI-express bus. Just as a practical example, running with 8 SMT threads and doing 100 evaluations of the entire model with 10 000 events takes in this case ~ 0.044 s. When runtimes become as low as that, transferring data over the PCI-express bus a 100 times can be the performance bottleneck because of the bus latency. In general, this suggests that the GPU should be used for a sufficient number of events, and this is suitable since the need for computing power increases as N increase. Luckily, we just depend on one number, so saturating bus bandwidth will never be an issue.

A significant improvement of the whole routine is the parallel reduction on the GPU. We clearly see that when doing parallel reduction with the CPU, the GPU is totally bound by it. Then loop fusions and constant expressions give significant speedups, at least together. The final tuning is to adjust workgroup sizes according to the "heuristic" we mentioned in Section 4.4. We emphasize that the runs without the occupancy optimizations was done with submitting NULL as a workgroup size to the runtime. According to the OpenCL standard the runtime should then choose the workgroup size it assumes would be the best.

Figure 4.7 shows the results for the comparison between the CPU and both GPUs on the same benchmark, but without 4 000 000 events because of a memory limit on the HD5870. The two GPUs clearly perform almost equally in most cases. Having the results from Section 4.3 in mind, this must mean that the computation is highly memory-bound. This seems reasonable, since the eta'K model involves a substantial fraction of composite PDFs, representing addition and multiplication. These operations are very fast and the memory accesses in these PDFs will clearly be a bottleneck. Just to illustrate this further, we give in Figure 4.8 the same benchmark, only with just the Gaussian function instead of the whole model. The results are revealing. The speedup of the HD5870 compared to the GTX470 is higher, which means a higher arithmetic intensity on the GPU, and the reason for this is that the Gaussian kernel itself represents a computation involving higher arithmetic intensity than what the eta'K model represents in total. In addition, less time is spent on the CPU since we are just calling kernels and not running any of the overhead associated with e.g. composite PDFs (in other words, a full model takes more CPU time to evaluate than one single function).

We think this gives a good explanation to the state of the application, and the main conclusion of these results is that the more complex model (amortizing bus latency) , the more events (amortizing CPU time), and, most importantly, the more computationally expensive kernels (amortizing memory access time), the better reason to utilize a GPU. We can in general say that both GPUs (and especially the HD5870) is severly under-utilized running this model, because of the bottleneck of global memory.

Timing is done just for evaluation, i.e. we assume that the overhead with setting up the

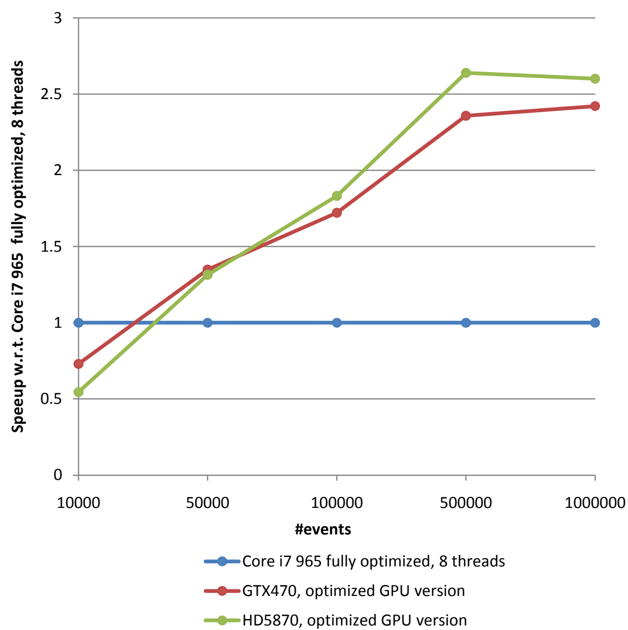


Figure 4.7: MLFit benchmark, comparison between CPU and both GPUs.

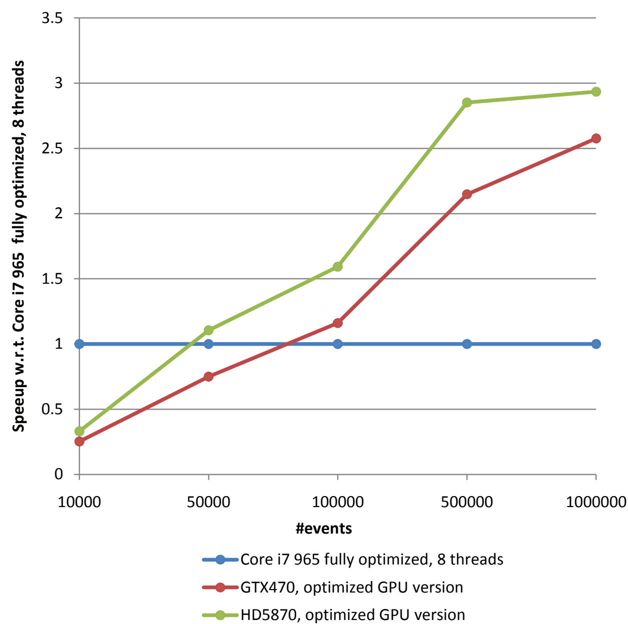


Figure 4.8: Gaussian function evaluation, CPU and both GPUs.

	$t_1(s)$	$t_2(s)$	$t_3(s)$	$t_4(s)$	$t_5(s)$	$\bar{t}$	σ	SE
GTX470	1.3282	1.3293	1.3287	1.3283	1.3276	1.3284	0.00063	0.028%
HD5870	1.1238	1.1211	1.113	1.1207	1.1121	1.118	0.0042	0.187%

Table 4.3: Timings, mean, standard deviation and standard error for 100 *NLL* evaluations with 1 000 000 events, both for the GTX470 and the HD5870.

OpenCL runtime and copying data from host to the GPU is amortized. The timing is done in the same way as when using the CPU, i.e. with the *omp_get_wtime* call surrounding the calls to the *NLL* function. This means that we take the total time also including copying the data over the PCI-Express bus. Thus, the timings are fair. Table 4.3 shows timings, mean, standard deviation and standard error for 5 runs with 100 *NLL* evaluations for both cards. Both cards deliver accurate timings in comparison with the CPU timings in Table 3.1, and we can note that the GTX470 is exceptionally accurate.

Results for a professional NVIDIA Tesla C2050 GPU are reported in Appendix F. We have chosen to put this in a separate appendix since we are primarily concerned with commodity cards in these chapters.

Chapter 5

Heterogeneous load balancing on commodity machines

We have in the past chapters shown the developments and optimizations of MLFit both for CPUs and GPUs, and benchmarked these implementations on a machine with a modern multi-core CPU and two modern GPUs. Still, it is not possible to exploit this machine fully by running on both the CPU and the GPU or in a multi-GPU configuration. It is interesting to look at how one can exploit a heterogeneous architecture like this in a hopefully implementation-pleasant way. It is possible to program both CPUs and GPUs with OpenCL (and OpenCL contains facilities for creating tasks to execute on either type of device), but we have shown that OpenCL for CPUs will not give us the performance we are looking for. In addition, the way of optimized tree evaluation we have implemented (implicitly parallel) would not be possible with OpenCL kernel calls for the CPU inside each tree node. In this chapter we show, with OpenMP and OpenCL combined, how much overhead there are involved when running with a heterogeneous configuration, and if there is anything to win on using both a CPU and a GPU together in our case. We believe there can be much to win on a multi-GPU solution.

5.1 Load balancing

An ideal execution of a compute-intensive program inside one physical computer makes use of all computation units of the computer. Load balancing is a very fundamental and old problem, and there has been done a lot of research on it ([44], [45], [46] and [47] are some publications dealing with different load balancing scenarios within computer science). In our

case, however, we can simplify the concept considerably. We hereby define a *compute device* as some device capable of computation. It can be a CPU (the physical chip, with cores), a GPU, an FPGA or some other custom-made accelerator hardware. Load balancing is often divided into two different types, namely *static*, and *dynamic* [24]. A static load balancing will be to distribute work across compute devices before execution has started. This will of course involve a guessing of the distribution, since the actual time of each device's execution is not known a priori (of course, in a homogeneous scenario it can be *nearly* known if each device's execution time is highly deterministic). This is indeed what we did with OpenMP in Chapter 3, where we knew that the cores were homogeneous and that the computations in each partition were equivalent for all threads. However, modern computers as the one we have used until now can contain compute devices with different specifications and capabilities, and it can therefore not be assumed that an equal distribution of work would lead to the optimal execution time.

Dynamic load balancing tries to even the execution times by splitting the whole workload into smaller tasks, so that each device can consume task by task. If the tasks are of an appropriate size, the execution time would be fairly optimal assuming low overhead related to the scheduling of them and that there is enough work to do to justify the overhead of e.g. memory copies. In many applications tasks can be dependent of each other, which often will result in a dependency graph in the implementation. We have already visited this concept by the use of TBB in Chapter 3. Since our case is quite specific, we feel that it will be superfluous to write more about the concept of load balancing in general. The next section will go straight to the point and give our motivations and arguments for the strategy we want to use.

5.2 Strategy

We use a refining static load balancing for the evaluation of the likelihood function. The whole evaluation can be regarded as data-parallel, and therefore we split the whole domain into partitions so that each compute device is responsible for computing one and only one partition. A central concept we already have discussed is the importance of result determinism. It is *crucial* that each compute device has a fixed range before actual result computation starts, since different ranges in the same run would lead to different results when the final reduction is done. This means that the balancing must be an initial phase before a real evaluation is performed. We have therefore implemented the following strategy for a heterogeneous

data-parallel load-balancer:

- Start an initial balancing phase by assigning each compute device a range of length approximately N/k, where N is the total number of elements, and k is the number of compute devices. This distribution will most probably be highly sub-optimal if the computational capabilities of the devices differ.

- Iterate by timing each compute device and adjusting the partitioning based on these timings. Hopefully, this will converge within some time threshold, assuring a nearly optimal execution. We denote the balancing iterations with the letter j.

The benefits of this strategy is that the result is deterministic and that, given a large enough workload, overhead associated with the balancing will be amortized. The immediate downside is that the entire subsequent evaluation will be based on this phase. If conditions in the system change, e.g. false timings during the initial phase or external load on the hardware during evaluation, the balancing can be sub-optimal. We see this load balancing as both static and dynamic. It is static in the way it divides the computation domain into fixed partitions a priori, and dynamic in the way that it adjusts itself based on real timings.

A good example of work done in the field of heterogeneous programming and load balancing is OMPSs, described by Ferrer et al. in [48]. OMPSs is a compiler directive driven approach to heterogeneous programming, and is task-based, much like how OpenMP 3.0 deals with declaring tasks [49]. It is supposed to support both CPUs and GPUs as well as some other accelerators. The most important feature with OMPSs in our case is that memory management is implicit in the compiler directives. We have to admit that programming with OpenCL in large programs can lead to a lot of code just managing memory to and from devices. But we also believe that this can be a downside, since using compiler directives exclusively is a large abstraction. There are in essence two main areas that are problematic with task-based dynamic load balancing in our case:

- A central concept is that the *NLL* evaluation *can* be extremely fast, but called many, many times, e.g. when doing minimization. If a task-based dynamic balancing is to be used, this will involve overhead in *each* call. This can potentially ruin performance. If we use an adjusting static load balancing algorithm, it will converge sooner or later, and overhead will be zero after that.

- As we pointed out with TBB in chapter 3, a deterministic reduction is not possible unless a fixed-size partition is assigned to each device.

The last point is the primary reason why we have excluded task-based dynamic load balancing as a potential solution for our scenario.

5.2.1 Method

The actual method used for dividing work between compute devices based on their respective execution times is described in this section. It is as far as we know originally described by Galindo et al. in [50] and is also used with seemingly good results in [51]. It could be regarded as reasonably simple and straight-forward, and should be optimal if consecutive timings have a small deviation/error. More formally this could be expressed as a series of timing values, where $t_1, t_2, ..., t_n$ are the timings of each device. Our goal is then to make a work partitioning that minimizes the difference between all these timing values within some threshold. Let n_i denote the partition for compute device i (i.e. the number of elements for that device), and let t_i denote its execution time. The balancing iterations are denoted with the letter j. We then define the *relative power* of this device (i.e. its relative compute capability)

$$RP_i^j = \frac{n_i^j}{t_i^j}, 0 \leq i < k, 0 \leq j \tag{5.2.1}$$

and the total power as

$$SRP^j = \sum_{i=0}^{k-1} RP_i^j. \tag{5.2.2}$$

Now each new partition can be computed by

$$n_i^{j+1} = N * \frac{RP_i^j}{SRP^j} \tag{5.2.3}$$

which hopefully will converge to an optimal distribution within some threshold as this routine is run over and over again.

5.2.2 Implementation details

First of all, it is important to mention the details about timings in this implementation. Timing was straightforward for both the CPU and the GPU in the previous chapters. However, when running in a hybrid scenario timing gets more challenging. The method described in Section 5.2.1 is implemented in a way similar to the conceptual implementation in listing 5.1. An essential part of the balancer is the function *timeComputeDeviceExecution*. However, we

cannot look at the CPU and the GPU as independent devices anymore. To execute kernel calls and copy data from the GPU to the host, the CPU must be involved. Before going into more details about the timing, we describe how threading is implemented in this scenario.

We mentioned in the beginning of the chapter that we wanted to see how OpenMP and OpenCL could work together, and we have tried to make the implementation as simple as possible. The way of threading is briefly illustrated in listing 5.2. First of all the number of threads set by the user is retrieved. We then aim to spawn one extra OpenMP thread for each GPU in the system, and this is made possible by the *omp_set_num_threads* call in the OpenMP API. We mean that this is a very simple and overcoming implementation since the OpenMP standard guarantees an implicit synchronization at the end of an OpenMP parallel region. It is important to remember that the OpenCL kernel calls, as we mentioned, are supposed to be non-blocking and ideally consume minimal CPU time. If this holds, we believe this implementation has the potential to be fast in a hybrid scenario (CPU + GPU(s)), i.e. that the evaluation of the tree for one or more GPU(s) would impose a minimal overhead. It will also be important that the GPU has enough work to do. If the kernel execution time is small, the fraction of the time the thread responsible for the GPU execution spends on GPU work will decrease while the fraction of the time spent on the CPU (traversing the tree) will increase, probably leading to a decreased efficiency since the CPU already is fully occupied doing computations.

Listing 5.1: A conceptual implementation of the balancing method.

```
double tmin = 1.0;
double tmax = 2.0;

while(tmax/tmin > threshold)
{
  for(int i = 0; i < k; i++)
  {
    t[i] = timeComputeDeviceExecution(i);
    if(t[i] > tmax) tmax = t[i];
    if(t[i] < tmin) tmin = t[i];
    RP[i] = n[i]/t[i];
    ...
  }
}
```

The obvious challenge here is to get an accurate timing for the execution on the CPU. There are two potential strategies:

- Since we use SPMD branching to manage what kind of work each thread should do, it is feasible to time the execution of the CPU threads only. This will ignore the eventual overhead the thread(s) used for managing one or more GPUs incur, so it will in essence be a heuristic.
- Time the full execution of the CPU threads and the extra thread(s) used for managing one or more GPUs. After all, this will be the true time the CPU will use. A downside with this is that the CPU is bound by the GPU, i.e. if the GPU is slower than the CPU (e.g. for very low workloads) the CPU timing will be false.

Both these approaches have been tried, and the former has in general been more successful. We have experienced larger timing variations and poorer results with the latter, which atleast partly can be explained by the fact that one timing error per device contributes to a total error when doing the timing. In general, one can say that the former should work well when the thread(s) used for managing one or more GPUs impose a low impact on the CPU. In essence this means that to get a good result from balancing, the overhead associated with the API calls to the OpenCL implementation must be low, and ideally as close to zero as possible. We have used the *omp_get_wtime* function for timing also in this scenario.

Listing 5.2: Threading implementation utilizing both CPU(s) and GPU(s).

```
int OMP_NUM_THREADS = omp_get_max_threads(); //Set by user as environmental variable
int numGPUs = getNumberOfGPUs();
omp_set_num_threads(OMP_NUM_THREADS + numGPUs);
#pragma omp parallel
{
  int threadID = omp_get_thread_num();
  if( <threadID corresponds to a GPU> )
    GetValGPU(threadID);
  else
    GetValCPU(); //Run CPU evaluation with all other threads
  //Implicit synchronization at the end of the region
}
```

Another important point is the convergence threshold. We have used a threshold of 1.03, which means that the timing between the fastest and the slowest device must be less or equal to 3%. This is a fair threshold since the errors for each device are lower (see Table 3.1 and 4.3). We have also called the evaluation within the *timeComputeDeviceExecution* function 5 times per measurement to get an accurate timing. An interesting observation is that the algorithm can go into an infinite loop. One can imagine that a workload is split between

two devices, and that the threshold is not reached. If we assume that the time of evaluating *one* element in the domain is so large that it will not get the ratio between the two devices any closer, it means that one of the devices will either give or receive one element of the domain from or to the other, and continue this way forever. But this can also happen for more than one element if there are large variations in timings. We have implemented a way of preventing this by increasing the threshold if convergence is not reached within a certain amount of iterations. However, in the case where the two devices exchange one element back and forth, this method can be slow, depending of how far one is from reaching the threshold. A faster method of preventing this is to set the threshold so that it takes into account the processing time of one single element at least, since this is our finest granularity. We have not applied this faster method since the time to process one event in our case is very fast. But it is worth to mention for other workloads, e.g. a very computationally expensive model which increases the processing time per event.

5.3 Benchmark results

We have tested the balancing feature on three use cases; two CPU-GPU scenarios and one multi-GPU scenario. The benchmark is the same as in the previous chapters. The following sections will describe the results specifically for each case. In the results we have a concept called *perfect speedup from load balancing*. This in essence means all speedups relative to the CPU execution timing. In other words, the CPU will always contribute with a speedup of 1, while a GPU would contribute with its own speedup compared to the CPU. In the multi-GPU scenario the perfect speedup is the relative speedup of both GPUs compared to the CPU, added together.

5.3.1 Test case 1: Intel Core i7 965 + NVIDIA GTX470

The event range used spans from 10 000 events to 4 000 000 events. Timings on the NVIDIA GTX470 have been incredibly accurate/stable, so the balancer have worked really well. An illustration of the balancing convergence on a run with 1 000 000 events is shown in Figure 5.1. The balancing converges in 3 steps, which means the timings are really reliable. Table 5.2 shows timing details. The standard error of the timings is here 0.57%.

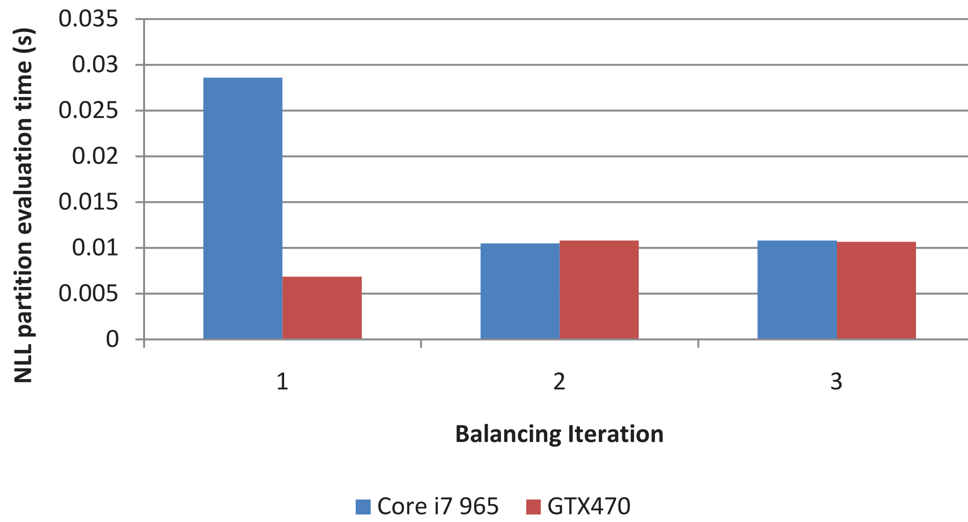


Figure 5.1: The balancing convergence of the Core i7 965 and the GTX470 together, running with 1 000 000 events.

$t_1(s)$	$t_2(s)$	$t_3(s)$	$t_4(s)$	$t_5(s)$	$\bar{t}$	σ	SE
1.0303	1.0294	1.0188	1.0464	1.0126	1.0275	0.0129	0.57%

Table 5.1: 5 timings, mean, standard deviation and standard error for the balanced run with the Core i7 965 and the GTX470. 1 000 000 events.

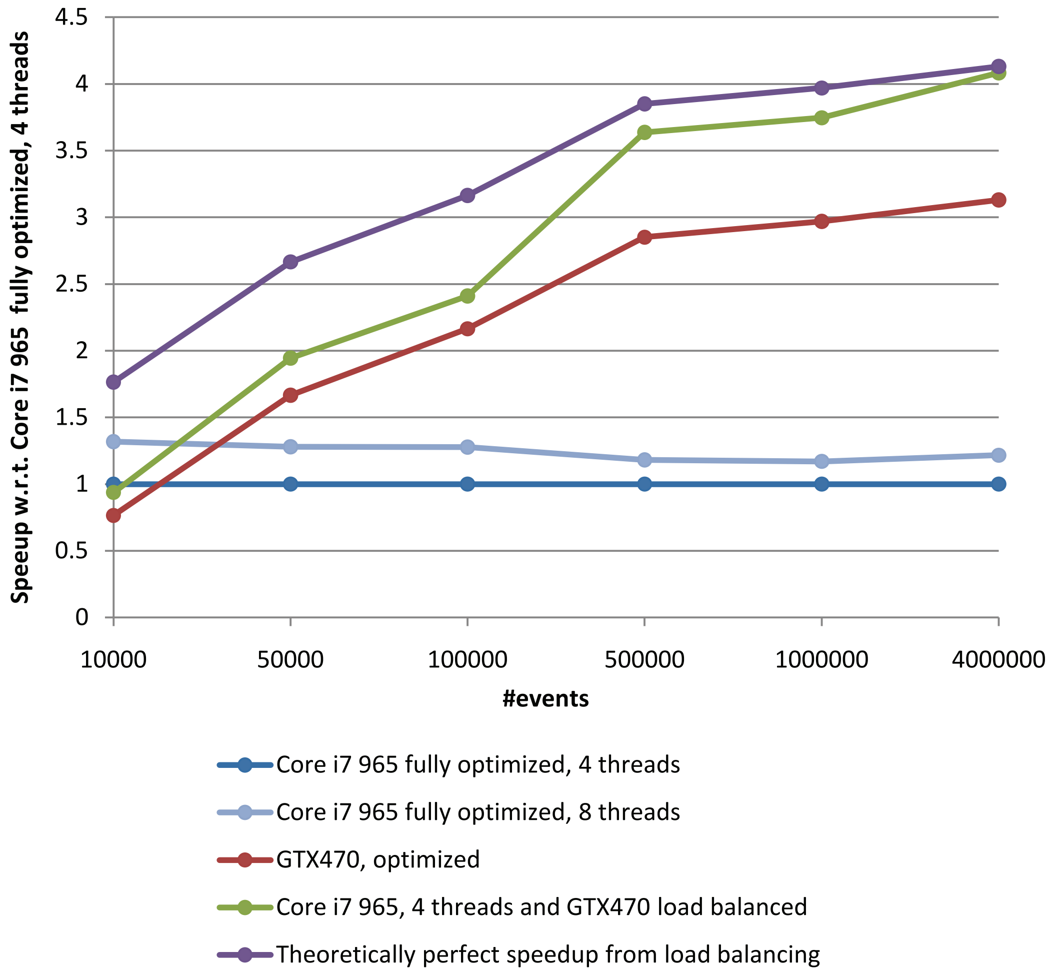


Figure 5.2: Speedup results from balancing between the Core i7 965 and the GTX470.

The speedup results are shown in Figure 5.2. First of all, we must add that there was no benefit in running with SMT anymore when involving an extra thread running the GPU execution. This might have to do with the overhead of running 9 threads instead of 5, when one of the threads are strictly diverging from the others, but it is difficult to pinpoint what exactly is the reason for this. The ideal case would be optimal balancing independent of the number of events, since the workload can be small while the routine is called many, many times. Unfortunately this is not the case. For 10 000, 50 000 and even 100 000 events the balanced version is only slightly faster than running on just the GPU. According to the speedup graph for the GPU, it is in general computationally starved for both 10 000, 50 000 and 100 000 events. This can mean that the fraction of the time the GPU spends on actual

computation is small compared to the time spent by the extra thread traversing the tree and executing kernels, and also the penalty this thread incurs on the other ones doing actual computations on the CPU. This again will propagate and maybe imply a larger error when doing the actual evaluation. If the GPU thread incurs substantial overhead, the timings will be wrong to a certain degree, since they represent the timings of an execution free of overhead.

Another important point to mention is that when introducing the CPU as a worker in scenarios with low workload, it will take work from a non-saturated GPU. Now, the speedup of the GPU compared to the CPU for a high number of events is in general higher than for a low number of events. This means that it is in general extremely difficult (if not impossible) to reach theoretically perfect speedup, as long as the GPU is not saturated. *This is the* m*ost important observation of the plots in this and the following sections.*

On the other hand, as the GPU gets saturated the balanced version starts to climb up under the theoretical maximum, which prooves an optimal balancing. This is a direct consequence of the saturation and the fact that the PDF evaluations take up most of the time, and thereby the amortization of the overhead associated with the extra OpenMP thread. Based on this plot, a preliminary conclusion of this balancing technique is that escaping overhead is difficult when the number of events is low, which is an unfortunate fact having the "low workload, very many evaluations" scenario in mind. It is also important to mention that balancing between a CPU and a GPU only makes sense when they perform comparable. Load balancing in this case works nearly optimal only when the GTX470 is $\sim 3x$ faster than the Core i7 965, where it gives a total speedup of $\sim 1.33x$.

5.3.2 Test case 2: Intel Core i7 965 + AMD Radeon HD5870

The load balancing between the Core i7 965 and the GTX470 showed promising and near-optimal results, atleast when amortizing overhead by increasing workload. The balancing convergence for the Core i7 965 and the HD5870 can be seen in Figure 5.3. Convergence takes longer time, and we belive this is a direct consequence of the larger error when timing the HD5870, according to Table 4.3. Table 5.2 shows timing details for this combination. The standard error is 1.83%, almost four times as large as when balancing with the GTX470.

$t_1(s)$	$t_2(s)$	$t_3(s)$	$t_4(s)$	$t_5(s)$	$\bar{t}$	σ	SE
1.033	1.016	1.1157	1.019	1.0374	1.04422	0.0409	1.83%

Table 5.2: 5 timings, mean, standard deviation and standard error for the balanced run with the Core i7 965 (3 threads) and the HD5870. 1 000 000 events.

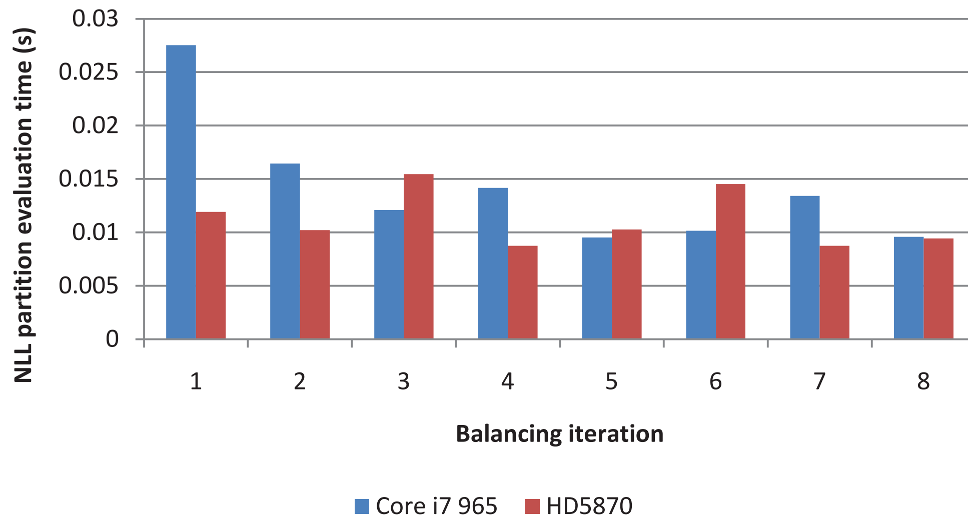


Figure 5.3: The balancing convergence of the Core i7 965 and the HD5870 together, running with 1 000 000 events.

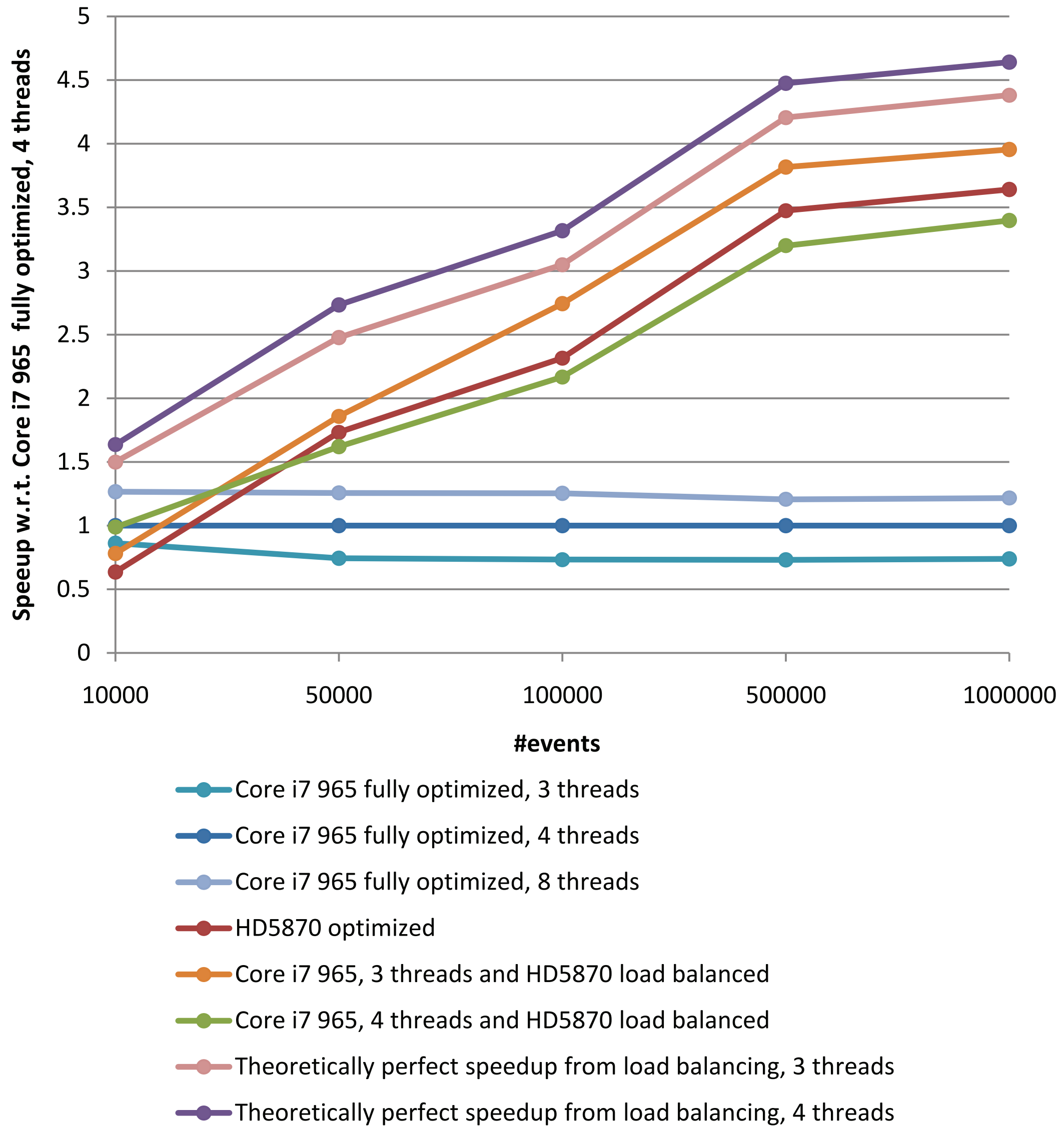


Figure 5.4: Speedup results from balancing between the Core i7 965 and the Radeon HD5870.

Speedup results are shown in Figure 5.4. The overhead associated with evaluating the tree and calling kernels must be more expensive for the CPU when running AMDs OpenCL implementation. We base this on the difference between the balanced execution with 3 threads and the one with 4 threads. Removing one thread obviously makes the balanced configuration beneficial, while running with 4 threads actually leads to performance loss compared to running evaluation with only the GPU. This is a clear sign of that thread number 5 has too much work to do, and disturbs the other 4 threads doing computation.

Figure 5.4 is different from Figure 5.2 in that the former does not include a run for 4 000

$t_1(s)$	$t_2(s)$	$t_3(s)$	$t_4(s)$	$t_5(s)$	$\bar{t}$	σ	SE
0.6635	0.6888	0.6625	0.7592	0.6452	0.6838	0.0449	2.01%

Table 5.3: 5 timings, mean, standard deviation and standard error for the balanced run with the GTX470 and the HD5870. 1 000 000 events.

000 events. This has to do with memory restrictions (not enough space) on the HD5870. It would be possible to run with 2 000 000 events, but we have experienced extremely unreliable timings with the HD5870 for a that high number of events. This can be a problem with the SDK implementation, the driver or the physical card. That said, which level the problem lies in is not interesting to us. It is more interesting to see how sensitive the balancer is to inaccurate timings.

Now, if we compare Figure 5.2 and Figure 5.4 in the case of 500 000 events, we can see that the GTX470 is almost perfectly balanced with the CPU when running with 4 computational CPU threads, while the same cannot be said about the HD5870 when running with just 3. We believe this is because the HD5870 initially is faster than the GTX470, and in addition to that, it is even faster when running with just 3 CPU threads. Also, load balancing will lead to a higher saturation point for the GPU as we mentioned in the last section (which is the most important observation also for this plot). Unfortunately, this makes the benefit of the hybrid solution rather small in the AMD case ($\sim 1.11x$ speedup for large workloads).

5.3.3 Test case 3: NVIDIA GTX470 + AMD Radeon HD5870

Our final test case is a multi-GPU scenario. We believe that this scenario has the greatest potential, since it relieves the CPU from computation, which again should make any overhead negligible. We therefore expect close to theoretically perfect speedup in this case, atleast when N grows. Timing details are shown in Table 5.3. It is important to remember that the workload practically is split in half when both GPUs are used, and we believe that can be a reason for the relatively high error compared to running with just the HD5870 (which is least reliable timing-wise).

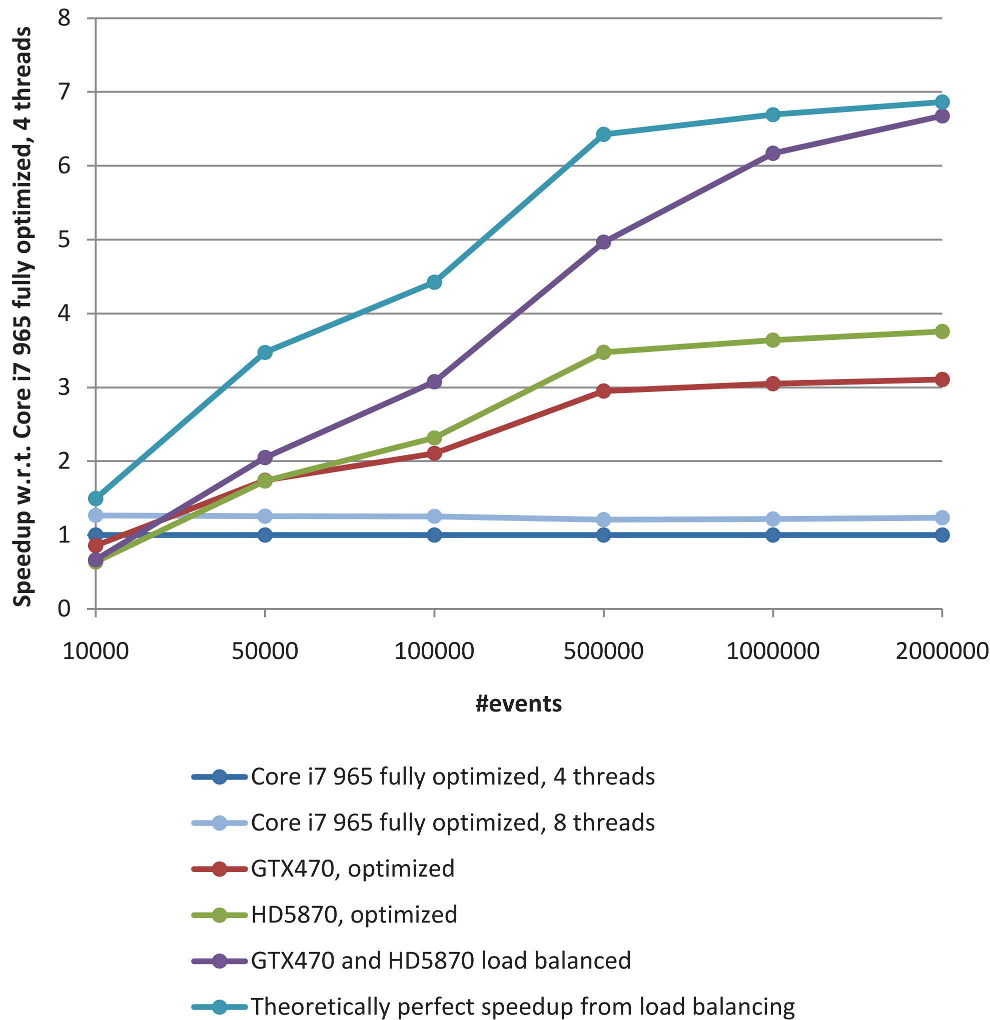


Figure 5.5: Speedup results from balancing between the GTX470 and the Radeon HD5870.

The results are shown in Figure 5.5. The principle of saturation point we mentioned is maybe most relevant in this case. The speedup on 1 000 000 events is comparable to the theoretical maximum at 500 000 events, suggesting quite similar load between the two cards, which is reasonable taking their similar performance into account. We can clearly see that the total climbs up under the theoretical maximum at 2 000 000 events in total as saturation is reached for both cards (2 000 000 events for two similar cards leads to approximately 1 000 000 events for each). Although the theoretical maximum is not reached for 500 000 events for instance, it clearly is beneficial to load balance, increasing speedup from $\sim 3.5x$ to $\sim 5x$. Running with a multi-GPU configuration is, at least for GPUs of this caliber, most ideal for very, very large workloads.

#devices (n)	n^2	$\sum_{r=1}^{n} \frac{n!}{r!(n-r)!}$	n^3
1	1	1	1
2	4	3	8
3	9	7	27
4	16	15	64
5	25	31	125
6	36	63	216
7	49	127	343

Table 5.4: A comparison between equation 5.4.2 and some well-known polynomials.

5.4 Considerations on optimal execution configurations

We want to discuss briefly how our implementation could decide the best execution configuration, i.e. which devices to use and eventually which not to use. As we have seen from many of the previous results, the choice of compute device(s) is important for performance. The most straight-forward and brute-force method of finding the optimal way of executing the fitting procedure is to enumerate all possible combinations of devices and pick the one that reveals the smallest timing. More formally, if we have three devices and denote each device with a D, then we want to enumerate all combinations, which in this case would be D_1, D_2, D_3, D_1D_2, D_1D_3, D_2D_3 and $D_1D_2D_3$. The formula for how many ways to pick r elements out of a population of n elements when the order is irrelevant is according to [52]

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}. \tag{5.4.1}$$

Now, since we do not know for sure that running with three devices will necessarily be faster than running with two (or one for that matter), the number of combinations needed to be tested is

$$\sum_{r=1}^{n} \frac{n!}{r!(n-r)!} \tag{5.4.2}$$

which obviously is computationally expensive, since the runtime of each combination must be obtained in addition. Table 5.4 compares the function value of this function with respect to the numer of devices, against some well-known polynomials. The complexity is not good, but it can be reasonable in cases where few devices are used. Its strength is that it guarantees an optimal running configuration, given that timings are accurate/reliable. An algorithm for generating these combinations can be written naively, or alternatively faster approaches are

described detailedly in [53]. The point is that this method must never be used for many devices.

Another approach which can be used for a higher amount of devices and can be classified as some sort of heuristic is the following:

- Time each device and start with the device with the lowest timing.
- Successively add the next best device to the configuration.
- If adding a device to a configuration leads to an execution with a higher timing, discard that device and proceed to the next (or eventually discard all the other devices, since they are assumed to slow down the execution).

This method is greedy and faster, with an approximated complexity of just $O(2n) \equiv O(n)$. However, these are just suggestions. The balancing feature should be tried by users in practice before any of these methods should be taken into consideration. Anything else would be premature engineering.

Chapter 6

Conclusions and future work

Our work has involved working closely with both CPUs and GPUs and we feel that there are some clear conclusions to draw. It has been a very practical thesis in many ways, but we believe there are quite a few findings that are just as relevant in general high-level software development geared towards performance.

6.1 Conclusions

First of all, this has been an interesting software package to work with, since it was non-trivial to make it fast and scalable. This is because all the high-level mechanisms from C++ that is used, and this reflects reality strongly. Most professional software is today written in either C or languages of a higher level, and thus we mean that this work can be interesting for other programmers dealing with forcing performance into high-level applications.

We started with some research questions and we feel we reached quite good answers to them after this work. The optimizations done in Chapter 3 made the whole application significantly faster while sacrifizing programmability to a certain degree. We estimated a single-core speedup of $\sim 7.8x$ compared to RooFit and we achieved a reasonably scalable application on a commodity processor ($\sim 3.6x$ on 4 cores, and $\sim 4.7x$ with 8 SMT threads).

Some of the most interesting findings was the experience with OpenCL. As long as the CPU implementations for OpenCL impose large overheads and lack of auto-vectorization (as well as lack of vectorization for transcendentals), they are really not attractive for professional use. The only benefit from using OpenCL for CPUs is code reuse and a somewhat unified programming model, and that could be a significant benefit for large software projects which aim to use both CPUs and GPUs for computation.

We predict that the programming of GPUs will remain as an offload-model for a long time still, atleast for applications geared towards maximum performance. OpenMP is supported by all main compilers in the market, and it works really well. As immature as OpenCL is at the time being, the only rational approach in a professional software setting where raw speed is required is to use plain C/C++, compile it in an optimized way, and use OpenCL kernels (or CUDA) as offload functions for accelerators. We also saw that OpenCL kernels for the CPU can be unsuitable in high-level applications cf. the implicitly parallel evaluation of the PDF tree/model when optimizations are to be done, because of the lack of C++ support. That said, we must admit that this has been a real stress test of OpenCL.

Chapter 4 revealed some interesting results regarding the GPUs in the market at the time being. While AMD and NVIDIA show off the theoretical peaks of their GPUs, the arithmetic intensity of a kernel must be *enormous* to exploit them, since global memory access brings enormous latencies. But clearly, if an application has intensive number crunching kernels with few memory operations, AMD GPUs are superior at the commodity level at the time being, if high-level programming constructs as e.g. polymorphism is not needed. We saw that combining programs written in a high-level way with GPUs can be difficult, as well as doing significant optimizations. Ideally, for maximum performance, users of RooFit should write their function in one kernel (compressing all functions into one), but then the entire point with the application is gone. Of course, for experiments where data analysis time can be very large, this could be the most pragmatic way of doing it. Then one very complex kernel can be load-balanced between a set of commodity GPUs if necessary, or run on a multi-core CPUs if the number of events is low.

The implementation of the load balancing uses a quite straightforward method of balancing. This is because we strongly believe it is the only rational approach in this case. Initially it is a trivial method, but it soon becomes more complex when taking timings into account. The user has two options; with p CPU cores he can use either p or $p - 1$ computational CPU threads, in addition to threads managing GPU execution. With p computational CPU threads, the only flaw with the implementation is that the timing of the CPU execution does not take into account the overhead of the threads handling the GPU execution, because timing the entire routine was highly unstable. We did not find any good solution to this, since it in essence was out of our control. The solution was to use a heuristic, i.e. the timing of the CPU evaluation only. The heuristic will work very well if the OpenCL implementation

- incurs low overhead when its API functions are called.
- provides stable and accurate timings. This of course also has to do with the whole

chain; SDK/implementation, maybe operating system, driver and physical card.

If these requirements are not met, $p-1$ computational CPU threads should be used. In the case of a quad-core processor this can mean a significant loss of power, but the number of cores increase fast by the time this is written, so for e.g. a 10-core processor this effect should be relatively small. We showed, with the NVIDIA GTX470 GPU and NVIDIAs OpenCL implementation, that the balancer (running with p computational CPU threads) will be optimal as long as these requirements are fulfilled and workload is high, saturating the GPU. We mean that for this application, the main conclusions of doing load balancing on a heterogeneous commodity machine with this kind of software are:

- Each compute device should perform comparable. If the first device is 5 times faster than the second then the overhead bound to balancing could make the final solution not beneficial. This was something we treated as a necessary pre-condition to even compare the CPU and the GPU, and it turned out that they were directly (1:1) comparable in some cases, but unfortunately these cases were not ideal cases for load balancing.

- If the CPU does computation with some threads itself, then all threads managing GPUs should impose an overhead close to zero. If not, the computational CPU threads can be disturbed, and the consequence may be sub-optimal runtimes compared to running with just one of the devices.

Another conclusion is that frameworks like OMPSs are not necessarily suitable for all kinds of parallel applications. In some scenarios, balancing must be specifically implemented to fulfill the requirements of a program.

If one takes the step from theory to what might actually be done, we think that users that has large computational jobs in front of them would be best of by utilizing a set of GPUs on a commodity machine. As Chapter 5 showed, load balancing between GPUs will be ideal as long as there is enough work to do, i.e. if kernels are very complex or the number of events is large (both fulfilled would be optimal). Our results also show that using a fast CPU (like the Intel Core i7 965) utilizing vectorization and all the cores is highly ideal when the number of events is low, so the methods of choosing optimal execution configurations can be relevant. If we compare the balanced execution with the GTX470 and the HD5870, it is almost 200x faster than running the old RooFit with a single core ($\sim 7.8 * \sim 3.6 * \sim 7 \approx 196$), just to give readers an idea of the potential with respect to the situation the way it is implemented today (and this can be achieved for roughly twice the price).

6.2 Future work

There are still some problems with MINUIT regarding minimization on GPUs bound to its strict requirements for accuracy. We have not looked into this in this work, but rather focused on the evaluation part of the likelihood. Future work will involve investigating how MINUIT can be changed or recompiled to allow usage of other devices than the CPU for minimization.

Apart from that we mean that most of the work with this application for commodity machines is done (i.e. on the node level). By that we mean that we do not see any other optimizations that can be done without breaking the interface and conventions RooFit consists of. A remaining work is of course to implement the methods used in MLFit, the prototype, into the main branch RooFit.

By our architectural optimizations, we hope that the performance characteristics will persist also when changing the physical model. However, this must be tested. Real experiments should use this version and provide feedback on how it works in practice for other models too.

Bibliography

[1] Roger Barlow. Introduction to statistical issues in particle physics, 2003. URL http://www.citebase.org/abstract?id=oai:arXiv.org:physics/0311105.

[2] Cern. http://cern.ch. Accessed 18.07.2011.

[3] Gordon Kane, editor. *Perspectives on LHC Physics*. World Scientific Publishing Company, 1 edition, June 2008. ISBN 9812833897. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/9812833897.

[4] S. Weinberg Phys. Rev. Lett. **19** 1264 (1967) S. L. Glashow, Nucl. Phys. **B22** 579 (1961) and A. Salam. *Elementary Particle Theory, ed. N. Svartholm, (Almqvist and Wiksell, Stockholm, 1968) p. 367.*

[5] The atlas experiment. http://atlas.ch. Accessed 31.07.2011.

[6] Cern openlab. https://proj-openlab-datagrid-public.web.cern.ch/proj-openlab-datagrid-public. Accessed 18.07.2011.

[7] B. Aubert et al. b meson decays to charmless meson pairs containing η or η' mesons. *Phys. Rev. D*, 80(11):112002, Dec 2009. doi: 10.1103/PhysRevD.80.112002.

[8] Slac national accelerator laboratory. http://www.slac.stanford.edu/. Accessed 18.07.2011.

[9] Encyclopaedia Britannica Online. "barn". http://www.britannica.com/EBchecked/topic/53558/barn. Accessed 18.07.2011.

[10] M. Bona et al. SuperB: A High-Luminosity Asymmetric e^+e^- Super Flavor Factory. Conceptual Design Report. 2007.

[11] Michelangelo L. Mangano. The super-lhc. 2009. URL http://arxiv.org/abs/0910.0030. cite arxiv:0910.0030 Comment: To appear in Contemporary Physics.

[12] W. Verkerke and D. Kirkby. The roofit toolkit for data modeling. proceedings of PHYSTAT05. Imperial College Press, London, 2006.

[13] R. Brun and F. Rademakers. Root - an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A*, 389:81, 1997.

[14] H. Albrecht et al. (argus collaboration), Nucl. Phys. **B241**, 278 (1990).

[15] F. James. *Statistical methods in Experimental Physics*. 2. edition.

[16] S. Jarp, A. Lazzaro, J. Leduc, A. Nowak, and F. Pantaleo. Evaluation of likelihood functions for data analysis on graphics processing units. http://cdsweb.cern.ch/record/1345075/, 2010.

[17] G. Cowan. *Statistical Data Analysis*. Clarendon Press, Oxford, 1998.

[18] F. James. In *MINUIT - Function Minimization and Error Analysis*, CERN Program Library Long Writeup D506, 1972.

[19] W. C. Davidon. Variable metric method for minimization. *SIAM J. Optim.*, 1:1–17, 1991.

[20] R. Fletcher. Comput. j. 13, (1970) 317.

[21] Alfio Lazzaro and Lorenzo Moneta. 2010 j. phys.: Conf. ser. Nucl. Phys. **B219** 042044.

[22] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[23] S. Jarp, A. Lazzaro, J. Leduc, A. Nowak, and F. Pantaleo. Parallelization of maximum likelihood fits with openmp and cuda. http://cdsweb.cern.ch/record/1328927, 2011.

[24] Barry Wilkinson and Michael Allen. *Parallel programming - techniques and applications using networked workstations and parallel computers (2. ed.)*. Pearson Education, 2005.

[25] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981. ISBN 0-201-03822-6.

[26] Peter Kornerup, Vincent Lefevre, Nicolas Louvet, and Jean-Michel Muller. On the computation of correctly-rounded sums. *IEEE Transactions on Computers*, 99(PrePrints), 2011. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2011.27.

[27] Peter Markstein. The new ieee-754 standard for floating point arithmetic. In Annie Cuyt, Walter Krämer, Wolfram Luther, and Peter Markstein, editors, *Numerical Validation in Current Hardware Architectures*, number 08021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. URL `http://drops.dagstuhl.de/opus/volltexte/2008/1448`.

[28] Benjamin C Allanach, D Grellscheid, and Fernando Quevedo. Genetic algorithms and experimental discrimination of susy models. *J. High Energy Phys.*, 07(hep-ph/0406277. DAMTP-2003-142):069. 23 p, Jun 2004.

[29] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18:1–33, January 2010. ISSN 1058-9244. URL `http://portal.acm.org/citation.cfm?id=1804799.1804800`.

[30] Khronos Group. Khronos group, open standards for media authoring and acceleration. `http://khronos.org`. Accessed 16.07.2011.

[31] NVIDIA Corporation. Opencl programming guide for the cuda architecture, version 2.3. `http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf`, . Accessed 10.07.2011.

[32] Advanced Micro Devices Inc. Amd app sdk. `http://developer.amd.com/sdks/amdappsdk/pages/default.aspx`, . Accessed 02.04.2011.

[33] Intel Corporation. Intel avx. `http://software.intel.com/en-us/avx/`, . Accessed 30.06.2011.

[34] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[35] Stuart F. Oberman, Student Member, and Michael J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46:154–161, 1997.

[36] Deborah T. Marr, Frank Binns, David L. Hill, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. 2002.

[37] Intel Corporation. 2nd generation intel(r) core(tm) processor family desktop and intel(r) pentium(r) processor(r) family desktop. http://download.intel.com/design/processor/datashts/324641.pdf, . Accessed 10.07.2011.

[38] NVIDIA Corporation. Geforce gtx 470. http://www.nvidia.com/object/product_geforce_gtx_470_us.html, . Accessed 10.07.2011.

[39] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. Languages and compilers for parallel computing. chapter MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pages 16–30. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-89739-2. doi: http://dx.doi.org/10.1007/978-3-540-89740-8_2. URL http://dx.doi.org/10.1007/978-3-540-89740-8_2.

[40] Y. Sneen Lindal. Parallelization of a state-of-the-art sph solver for water simulations on modern gpus. NTNU, 2010.

[41] Advanced Micro Devices Inc. Ati radeon(tm) hd 5870 graphics. http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-overview.aspx#2, . Accessed 10.07.2011.

[42] NVIDIA Corporation. Nvidias next generation cuda(tm) compute architecture: Fermi, version 1.1. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, . Accessed 10.07.2011.

[43] Mihai Niculescu and Sorin-Ion Zgura. Computing trends using graphic processor in high energy physics. June 2011. URL http://arxiv.org/abs/1106.6217.

[44] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279 – 301, 1989. ISSN 0743-7315. doi: DOI:10.1016/0743-7315(89)90021-X. URL http://www.sciencedirect.com/science/article/pii/074373158990021X.

[45] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15:253–285, August 1997. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/263326.263344. URL http://doi.acm.org/10.1145/263326.263344.

[46] V. Cardellini, M. Colajanni, and P.S. Yu. Dynamic load balancing on web-server systems. *Internet Computing, IEEE*, 3(3):28 –39, may/jun 1999. ISSN 1089-7801. doi: 10.1109/4236.769420.

[47] Behrooz A. Shirazi, Krishna M. Kavi, and Ali R. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995. ISBN 0818665874.

[48] Roger Ferrer, Judit Planas, Pieter Bellens, Alejandro Duran, Marc Gonzalez, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesus Labarta. Optimizing the exploitation of multicore processors and gpus with openmp and opencl. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 215–229, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19594-5. URL `http://portal.acm.org/citation.cfm?id=1964536.1964551`.

[49] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20:404–418, March 2009. ISSN 1045-9219. doi: 10.1109/TPDS.2008.105. URL `http://portal.acm.org/citation.cfm?id=1512157.1512430`.

[50] Ismael Galindo, Francisco Almeida, and Jose Manuel Badia-Contelles. Dynamic load balancing on dedicated heterogeneous systems. In Alexey L. Lastovetsky, M. Tahar Kechadi, and Jack Dongarra, editors, *PVM/MPI*, volume 5205 of *Lecture Notes in Computer Science*, pages 64–74. Springer, 2008. ISBN 978-3-540-87474-4. URL `http://dblp.uni-trier.de/db/conf/pvm/pvm2008.html#GalindoAB08`.

[51] Alejandro Acosta, Robert Corujo, Vicente Blanco Pérez, and Francisco Almeida. Dynamic load balancing on heterogeneous multicore/multigpu systems. In *HPCS*, pages 467–476, 2010.

[52] Eric R. Ziegel. Probability and statistics for engineering and the sciences (6th ed.), by jay l. devore. *Technometrics*, 46(4):497+. ISSN 0040-1706. URL `http://www.ingentaconnect.com/content/asa/tech/2004/00000046/00000004/art00034`.

[53] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley Professional, 2005. ISBN 0201853949.

[54] Tesla c2050 / c2070 gpu computing processor - supercomputing at 1/10th the cost. `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf`. Accessed 31.07.2011.

Appendix A

OpenCL test kernels with varying arithmetic intensity

Listing A.1: OpenCL kernels with varying arithmetic intensity

```
__kernel void kernel1(global double* a, global double* results, int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  double aval = a[i];
  double bval = aval*1.5;
  aval += aval*(bval+bval*(aval*bval*(bval+bval*(aval+aval))));
  results[i] = aval;
}

__kernel void kernel2(global double* a, global double* results, int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  double aval = a[i];
  double bval = aval*1.5;
  aval += aval*(bval+bval*(aval*bval*(bval+bval*(aval+aval)+bval*bval*(bval+aval-(aval*
      bval+aval*(bval-aval*(aval-bval)+aval))))));
  results[i] = aval;
}

__kernel void kernel3(global double* a, global double* results, int N)
{
  int i = get_global_id(0);
```

```
  if(i >= N) return;
  double aval = a[i];
  double bval = aval*1.5;
  for(int i = 0; i < 10; i++)
    aval += aval*(bval+bval*(aval*bval*(bval+bval*(aval+aval))));
  results[i] = aval;
}

__kernel void kernel4(global double* a, global double* results, int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  double aval = a[i];
  double bval = aval*1.5;
  for(int i = 0; i < 100; i++)
    aval += aval*(bval+bval*(aval*bval*(bval+bval*(aval+aval)+bval*bval*(bval+aval-(aval*
        bval+aval*(bval-aval*(aval-bval)+aval))))));
  results[i] = aval;
}

__kernel void kernel5(global double* a, global double* results, int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  double aval = a[i];
  double bval = aval*1.5;
  for(int i = 0; i < 1000; i++)
    aval += aval*(bval+bval*(aval*bval*(bval+bval*(aval+aval)+bval*bval*(bval+aval-(aval*
        bval+aval*(bval-aval*(aval-bval)+aval))))));
  results[i] = aval;
}

__kernel void kernel6(global double* a, global double* results, int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  double aval = a[i];
  double bval = aval*1.5;
  for(int i = 0; i < 10000; i++)
    aval += aval*(bval+bval*(aval*bval*(bval+bval*(aval+aval)+bval*bval*(bval+aval-(aval*
        bval+aval*(bval-aval*(aval-bval)+aval))))));
  results[i] = aval;
```

```
}
```

Appendix B

OpenCL test kernels involving transcendentals

Listing B.1: OpenCL test kernels involving transcendentals

```
__kernel void kernel1(global double* a, global double* b, global double* results, int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  results[i] = a[i] + b[i];
}

__kernel void kernel2(global double* a, global double* b, global double* results, int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  results[i] = 5.0*a[i] + b[i];
}

__kernel void kernel3(global double* a, global double* b, global double* results, int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  results[i] = 5.0*a[i] + exp(b[i]);
}

__kernel void kernel4(global double* a, global double* b, global double* results, int N)
{
  int i = get_global_id(0);
```

```
  if(i >= N) return;
  results[i] = 5.0*a[i] + log(b[i]);
}

__kernel void kernel5(global double* a, global double* b, global double* results, int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  double aval = a[i];
  double bval = b[i];
  results[i] = log(log(aval*2.5))/log(bval)*log(aval)*pow(aval, 3.5)*pow(aval, 3.2)*log(
      bval*2.1);
}

__kernel void kernel6(global double* a, global double* b, global double* results, int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  double aval = a[i];
  double bval = b[i];
  results[i] = log(log(aval*2.5))/log(bval)*log(aval)*pow(aval, 3.5)*pow(aval, 3.2)*log(
      bval*2.1)/log(pow(bval*aval, 3.4));
}
```

Appendix C

The eta'K model implementation

Listing C.1: The eta'K model implementation

```
// Data analysis of branching fraction measurement of eta'(rg) K0S (PhysRevD.80.112002)

#include "Variable.h"
#include "PdfGaussian.h"
#include "PdfProd.h"
#include "PdfAdd.h"
#include "PdfBreitWigner.h"
#include "PdfPolynomial.h"
#include "PdfBifurGaussian.h"
#include "PdfArgusBG.h"

// Define the model
RooAbsPdf *ModelEtapRGKs(RooRealVar &x, RooRealVar &y, RooRealVar &z,
      const Int_t N)
{
  // Var x

  RooRealVar *muA1x = new RooRealVar("muA1x","",-0.0018,-0.01,0.01);
  muA1x->setError(0.001);
  RooRealVar *sigmaA1x = new RooRealVar("sigmaA1x","",0.01512,0,0.1);
  sigmaA1x->setError(0.001);
  RooAbsPdf *gaussA1x = new RooGaussian("gaussA1x","",x,*muA1x,*sigmaA1x);

  RooRealVar *coeff1B1x = new RooRealVar("coeff1B1x","",-0.3156,-1,1);
  coeff1B1x->setError(0.001);
  List<Variable> coefficientsB1x(*coeff1B1x);
```

```
RooAbsPdf *polyB1x = new RooPolynomial("polyB1x","",x,coefficientsB1x);

RooRealVar *coeff1C1x = new RooRealVar("coeff1C1x","",-0.7728);
RooRealVar *coeff2C1x = new RooRealVar("coeff2C1x","",0.0067);
RooRealVar *coeff3C1x = new RooRealVar("coeff3C1x","",0.1047);
RooRealVar *coeff4C1x = new RooRealVar("coeff4C1x","",-0.1120);
List<Variable> coefficientsC1x;
coefficientsC1x.add(*coeff1C1x);
coefficientsC1x.add(*coeff2C1x);
coefficientsC1x.add(*coeff3C1x);
coefficientsC1x.add(*coeff4C1x);
RooAbsPdf *polyC1x = new RooPolynomial("polyC1x","",x,coefficientsC1x);

RooRealVar *muD1x = new RooRealVar("muD1x","",0.1115);
RooRealVar *sigmaD1x = new RooRealVar("sigmaD1x","",0.0464);
RooAbsPdf *gaussD1x = new RooGaussian("gaussD1x","",x,*muD1x,*sigmaD1x);
RooRealVar *coeff1D2x = new RooRealVar("coeff1D2x","",0.4146);
List<Variable> coefficientsD2x(*coeff1D2x);
RooAbsPdf *polyD2x = new RooPolynomial("polyD2x","",x,coefficientsD2x);
RooRealVar *fracDx = new RooRealVar("fracDx","",0.3821);
RooAbsPdf *addDx = new RooAddPdf("addDx","",*gaussD1x,*polyD2x,*fracDx);

RooRealVar *coeff1E1x = new RooRealVar("coeff1E1x","",-0.9392);
RooRealVar *coeff2E1x = new RooRealVar("coeff2E1x","",-0.0793);
RooRealVar *coeff3E1x = new RooRealVar("coeff3E1x","",0.2838);
RooRealVar *coeff4E1x = new RooRealVar("coeff4E1x","",-0.1428);
List<Variable> coefficientsE1x;
coefficientsE1x.add(*coeff1E1x);
coefficientsE1x.add(*coeff2E1x);
coefficientsE1x.add(*coeff3E1x);
coefficientsE1x.add(*coeff4E1x);
RooAbsPdf *polyE1x = new RooPolynomial("polyE1x","",x,coefficientsE1x);

// Var y

RooRealVar *muA1y = new RooRealVar("muA1y","",5.2798,5.27,5.29);
muA1y->setError(0.001);
RooRealVar *sigmaA1y = new RooRealVar("sigmaA1y","",0.002640,0,0.01);
sigmaA1y->setError(0.001);
RooAbsPdf *gaussA1y = new RooGaussian("gaussA1y","",y,*muA1y,*sigmaA1y);

RooRealVar *mB1y = new RooRealVar("mB1y","",y.getMax());
```

```
RooRealVar *cB1y = new RooRealVar("cB1y","",-27.8171,-40,-10);
cB1y->setError(0.001);
RooAbsPdf *argusB1y = new RooArgusBG("argusB1y","",y,*mB1y,*cB1y);

RooRealVar *mC1y = new RooRealVar("mC1y","",y.getMax());
RooRealVar *cC1y = new RooRealVar("cC1y","",-65.2194);
RooAbsPdf *argusC1y = new RooArgusBG("argusC1y","",y,*mC1y,*cC1y);
RooRealVar *muC2y = new RooRealVar("muC2y","",5.2808);
RooRealVar *sigmaC2y = new RooRealVar("sigmaC2y","",0.0041);
RooAbsPdf *gaussC2y = new RooGaussian("gaussC2y","",y,*muC2y,*sigmaC2y);
RooRealVar *fracCy = new RooRealVar("fracCy","",0.8576);
RooAbsPdf *addCy = new RooAddPdf("addCy","",*argusC1y,*gaussC2y,*fracCy);

RooRealVar *muD1y = new RooRealVar("muD1y","",5.2785);
RooRealVar *sigmaD1y = new RooRealVar("sigmaD1y","",0.0054);
RooAbsPdf *gaussD1y = new RooGaussian("gaussD1y","",y,*muD1y,*sigmaD1y);

RooRealVar *mE1y = new RooRealVar("mE1y","",y.getMax());
RooRealVar *cE1y = new RooRealVar("cE1y","",-61.2961);
RooAbsPdf *argusE1y = new RooArgusBG("argusE1y","",y,*mE1y,*cE1y);
RooRealVar *muE2y = new RooRealVar("muE2y","",5.2784);
RooRealVar *sigmaE2y = new RooRealVar("sigmaE2y","",0.0050);
RooAbsPdf *gaussE2y = new RooGaussian("gaussE2y","",y,*muE2y,*sigmaE2y);
RooRealVar *fracEy = new RooRealVar("fracEy","",0.6793);
RooAbsPdf *addEy = new RooAddPdf("addEy","",*argusE1y,*gaussE2y,*fracEy);

// Var z

RooRealVar *muA1z = new RooRealVar("muA1z","",-0.5518,-1,1);
muA1z->setError(0.001);
RooRealVar *sigmaA1z = new RooRealVar("sigmaA1z","",0.3314,0,1);
sigmaA1z->setError(0.001);
RooAbsPdf *gaussA1z = new RooGaussian("gaussA1z","",z,*muA1z,*sigmaA1z);

RooRealVar *muB1z = new RooRealVar("muB1z","",-1.1352,-1.5,-0.5);
muB1z->setError(0.001);
RooRealVar *sigmaLB1z = new RooRealVar("sigmaLB1z","",0.3321,0,1);
sigmaLB1z->setError(0.001);
RooRealVar *sigmaRB1z = new RooRealVar("sigmaRB1z","",0.4441,0,1);
sigmaRB1z->setError(0.001);
RooAbsPdf *bifurgaussB1z = new RooBifurGauss("bifurgaussB1z","",z,*muB1z,*sigmaLB1z,*
    sigmaRB1z);
```

```
RooAbsPdf *polyB1z = new RooPolynomial("polyB1z","",z);
RooRealVar *fracBz = new RooRealVar("fracBz","",0.99);
RooAbsPdf *addBz = new RooAddPdf("addBz","",*bifurgaussB1z,*polyB1z,*fracBz);

RooRealVar *muC1z = new RooRealVar("muC1z","",-0.6762);
RooRealVar *sigmaLC1z = new RooRealVar("sigmaLC1z","",0.3241);
RooRealVar *sigmaRC1z = new RooRealVar("sigmaRC1z","",0.3477);
RooAbsPdf *bifurgaussC1z = new RooBifurGauss("bifurgaussC1z","",z,*muC1z,*sigmaLC1z,*
    sigmaRC1z);

RooRealVar *muD1z = new RooRealVar("muD1z","",-0.6529);
RooRealVar *sigmaLD1z = new RooRealVar("sigmaLD1z","",0.3472);
RooRealVar *sigmaRD1z = new RooRealVar("sigmaRD1z","",0.3577);
RooAbsPdf *bifurgaussD1z = new RooBifurGauss("bifurgaussD1z","",z,*muD1z,*sigmaLD1z,*
    sigmaRD1z);

RooRealVar *muE1z = new RooRealVar("muE1z","",-0.6336);
RooRealVar *sigmaLE1z = new RooRealVar("sigmaLE1z","",0.3440);
RooRealVar *sigmaRE1z = new RooRealVar("sigmaRE1z","",0.3570);
RooAbsPdf *bifurgaussE1z = new RooBifurGauss("bifurgaussE1z","",z,*muE1z,*sigmaLE1z,*
    sigmaRE1z);


RooRealVar *nA = new RooRealVar("nA","",10,0,N); nA->setError(1);
RooRealVar *nB = new RooRealVar("nB","",40,0,N); nB->setError(1);
RooRealVar *nC = new RooRealVar("nC","",30,0,N); nC->setError(1);
RooRealVar *nD = new RooRealVar("nD","",10,0,N); nD->setError(1);
RooRealVar *nE = new RooRealVar("nE","",10,0,N); nE->setError(1);
List<Variable> nevents;
nevents.add(*nA); nevents.add(*nB); nevents.add(*nC); nevents.add(*nD); nevents.add(*nE)
    ;

RooAbsPdf *pdfA = new RooProdPdf("pdfA","",List<AbsPdf>(*gaussA1x,*gaussA1y,*gaussA1z));
RooAbsPdf *pdfB = new RooProdPdf("pdfB","",List<AbsPdf>(*polyB1x,*argusB1y,*addBz));
RooAbsPdf *pdfC = new RooProdPdf("pdfC","",List<AbsPdf>(*polyC1x,*addCy,*bifurgaussC1z))
    ;
RooAbsPdf *pdfD = new RooProdPdf("pdfD","",List<AbsPdf>(*addDx,*gaussD1y,*bifurgaussD1z)
    );
RooAbsPdf *pdfE = new RooProdPdf("pdfE","",List<AbsPdf>(*polyE1x,*addEy,*bifurgaussE1z))
    ;
List<AbsPdf> Pdfs;
Pdfs.add(*pdfA); Pdfs.add(*pdfB); Pdfs.add(*pdfC); Pdfs.add(*pdfD); Pdfs.add(*pdfE);
```

```
  return new RooAddPdf("extended","",Pdfs,nevents);
}
```

Appendix D

A tree illustration of the eta'K model

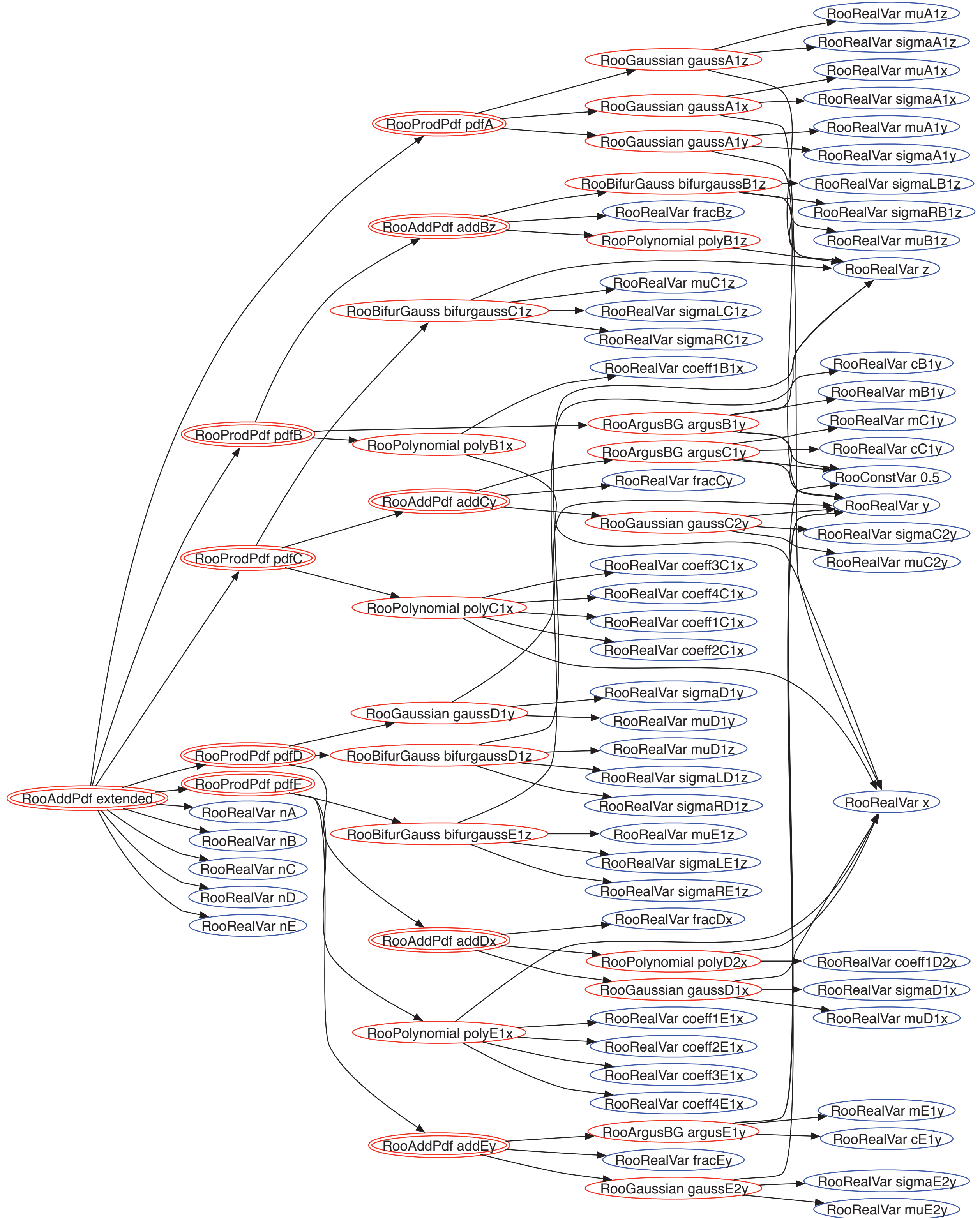


Figure D.1: Tree illustration of the eta'K model. Double-circled red nodes are composite PDFs, single-circled are ordinary PDFs and blue nodes are variables.

Appendix E

OpenCL kernels for the PDFs used in the eta'K model

Listing E.1: OpenCL kernels for the PDFs used in the eta'K model

```
/**
 * AbsPdf log value kernel
 */
 __kernel void logValue(__global double *results, const int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  results[i] = log(results[i]);
}



/**
 * AbsPdf normalization kernel
 */
__kernel void normalizeResults(const double invIntegral, __global double *results, const
    int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  results[i] *= invIntegral;
}



/**
```

```
 * PdfAdd kernel
 */
 __kernel void evaluatePdfAdd(__global double *data1, __global double *data2,
    double coeff1, double coeff2, __global double *results, const int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  results[i] = mad(coeff1, data1[i], coeff2 * data2[i]);
}


/**
 * PdfProd kernel
 */
__kernel void evaluatePdfProd(__global double *data1, __global double *data2,
    __global double *results, const int N)
{
  int i = get_global_id(0);
  if(i >= N) return;
  results[i] = data1[i] * data2[i];
}


/**
 * Argus BG evaluation kernel
 */
__kernel void evaluatePdfArgusBG(const double invm0, const double c, __global const
    double *data, const int variableOffset, __global double *results, const int N, const
    double invIntegral)
{
  int i = get_global_id(0);
  if(i >= N) return;
  const int offset = i + variableOffset;
  double d = data[offset];
  double t = d*invm0;
  if(t >= 1.)
  {
    results[i] = 0.0;
  }
  else
  {
    double u = ((double)1.) - t*t;
```

```
    results[i] = d*sqrt(u)*exp(c*u) * invIntegral;
  }
}


/**
 * Bifur Gaussian evaluation kernel
 */
 __kernel void evaluatePdfBifurGaussian(const double mu, const double coeff1, const double
      coeff2, __global const double *data, const int variableOffset, __global double *
     results, const int N, const double invIntegral)
{
  int i = get_global_id(0);
  if (i >= N) return;
  double x = data[i + variableOffset];
  double arg = x - mu;
  double coeff = 0.0;
  if (arg < 0.0)
  {
    coeff = coeff1;
  }
  else
  {
    coeff = coeff2;
  }
  results[i] = exp(coeff*arg*arg) * invIntegral;
}


/**
 * Gaussian evaluation kernel
 */
  __kernel void evaluatePdfGaussian(const double mu, const double sigma, __global const
      double *data, const int variableOffset, __global double *results, const int N, const
       double invIntegral)
{
  int i = get_global_id(0);
  if (i >= N) return;
  double x = data[i + variableOffset];
  double temp = (x-mu)/sigma;
  temp *= temp;
  results[i] = exp(-0.5*temp) * invIntegral;
```

```
}



/**
 * Polynomial evaluation kernel
 */
 __kernel void evaluatePdfPolynomial(__global const double *coeff, unsigned int ncoeff,
     __global const double *data, const int variableOffset, __global double *results,
     const int N, const double invIntegral)
{
  int i = get_global_id(0);
  if (i >= N) return;
  const int offset = i + variableOffset;
  double result = coeff[ncoeff];
  for (;ncoeff>0;--ncoeff)
    result = mad(result, data[offset], coeff[ncoeff-1]);
  results[i] = result * invIntegral;
}
```

Appendix F

NVIDIA Tesla benchmarks

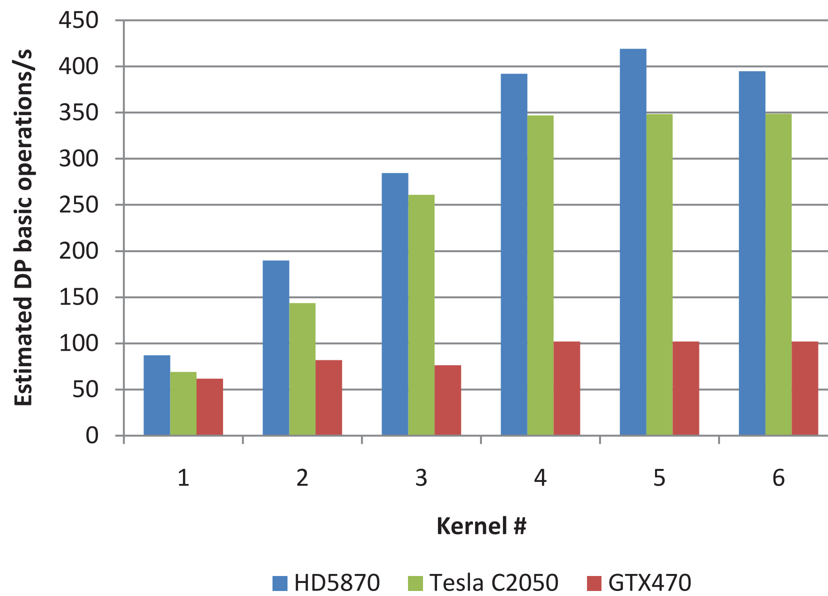
We want to briefly show a few results comparing an NVIDIA Tesla C2050 professional GPU to the NVIDIA GTX470 and the AMD Radeon HD5870 from chapter 4. The Tesla series from NVIDIA are directed towards professional use, and do not have the same restrictions of double-precision performance as e.g. the GTX gamer GPUs. Table F.1 shows the specifications of the Tesla C2050 GPU. These figures are taken from tes [54], which is a product sheet from NVIDIA. The core frequency is slighly lower than on the GTX470, and thereby the single-precision peak performance is also slightly lower. But the double-precision peak performance is approximately four times as high.

Recall the basic operation test we did in section 4.3. The same test including the Tesla C2050 is shown in figure F.1. The results speak for themselves. The theoretical difference in peak performance is clear for both double-precision and single-precision accuracy.

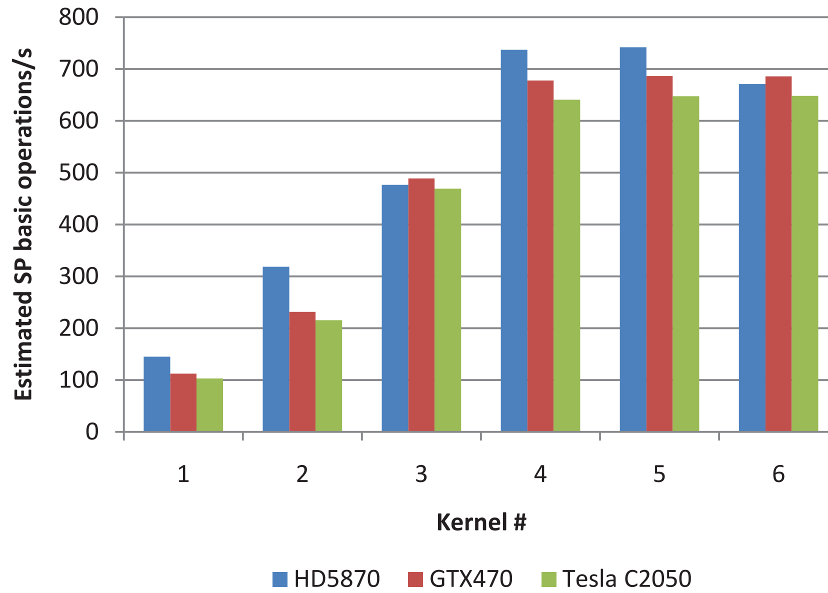
But, what happens when running with the transcendental kernels made to be more similar to the typical kernels in the eta'K model? Figure F.2 shows these results. The plot is identical to Figure 4.3 but with a linear vertical axis instead of logarithmic, in addition to including the Tesla results. In essence these results prooves that there is nothing to gain on a Tesla

	NVIDIA Tesla C2050
Number of ALUs/scalar cores	448
Core clock	1150 MHz
Peak single-precision performance	1030 GFLOP/s
Peak double-precision performance	515 GFLOP/s
Memory bandwidth	144 GB/s
Memory size	3 GB

Table F.1: The specifications of the NVIDIA Tesla C2050 professional GPU.



(a) Double-precision accuracy.



(b) Single-precision accuracy.

Figure F.1: Equivalent to figure 4.2, but with the Tesla C2050 in addition.

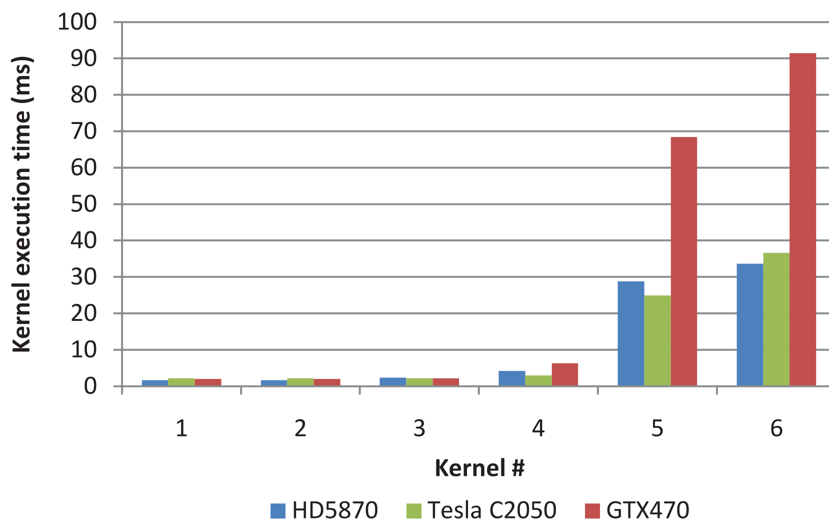


Figure F.2: Tesla results for the kernels involving transcendentals in Appendix B.

GPU if kernels are memory-bound. In essence, this should mean that the eta'K evaluation should perform fairly similar whether one runs with a GTX470 or with a Tesla, and this is indeed confirmed in Figure F.3. The difference when N grows high is most probably because of the higher memory bandwith of the Tesla. The conclusion is clear.

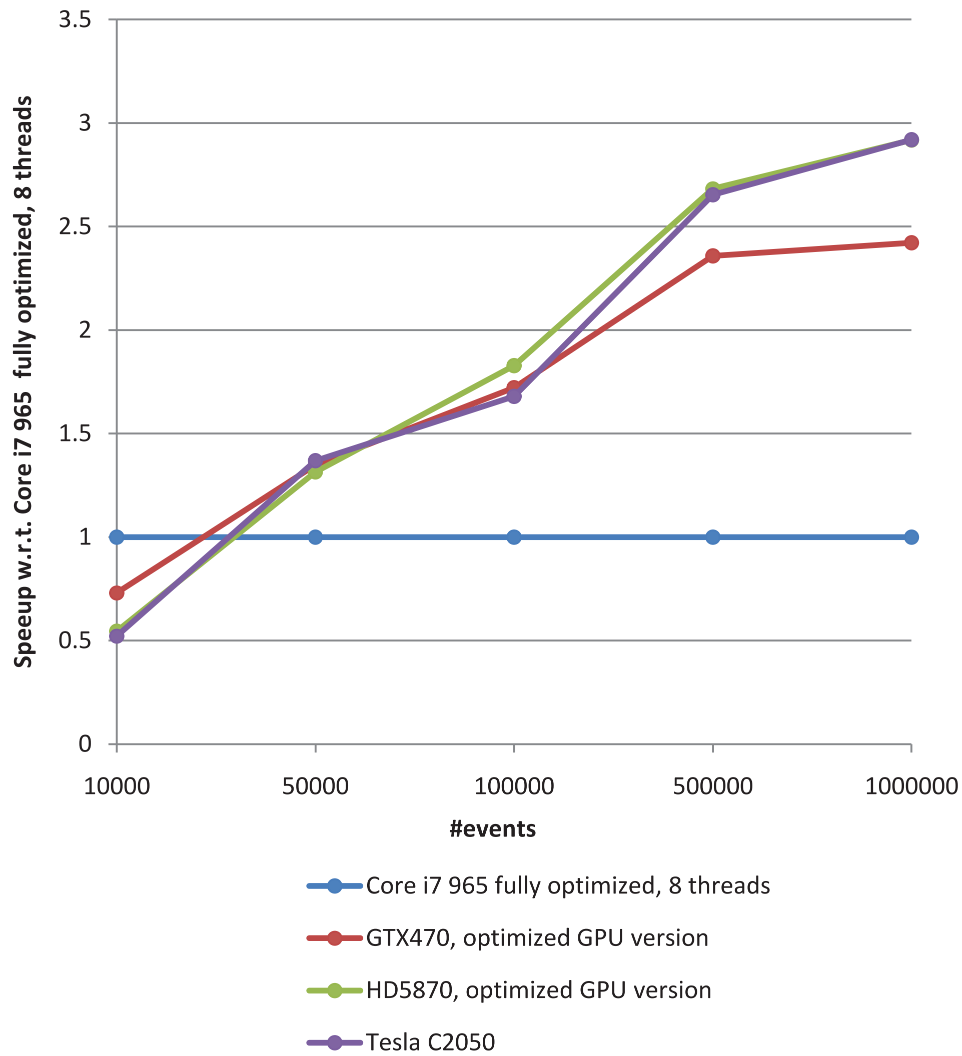


Figure F.3: MLFit benchmark including results for the Tesla C2050. Eta'K model evaluation.

Appendix G

CERN openlab report: *First encounter with OpenCL for multicore CPUs*

First encounter with OpenCL for multicore CPUs

Yngve Sneen Lindal, Sverre Jarp, Alfio Lazzaro, Julien Leduc, Andrzej Nowak

July 13, 2011

yngve.sneen.lindal@cern.ch

Contents

1 Introduction **3**
1.1 Maximum likelihood fits 3
1.2 NLL evaluation 4

2 *NLL* Evaluation with OpenCL **5**

3 Drawbacks of OpenCL on CPUs **6**
3.1 Explicit vectorization 7
3.2 Thread scheduling 8

4 Conclusion **9**

5 Intel OpenCL wish list **9**

1 Introduction

This short report is centered around the same application as described in [1]. Therefore, sections 1.1 and 1.2 are copied from [1], so that the reader can have a short introduction of the application we are working with. This application was parallelized using OpenMP and CUDA with good scalability and performance. We reached a speedup factor of 3.8x using OpenMP using 4 threads. We use this result as a reference for a new implementation in OpenCL which is described in the rest of this report. We aim to have a common implementation (equal kernels) of this application for the CPU and the GPU.

1.1 Maximum likelihood fits

The maximum likelihood (ML) fitting procedure is a popular statistical technique used to estimate parameters of a statistical model on a given data sample [2]. Data samples are a collection of N independent *events*, an event being the measurement of a set of *variables* $\hat{x} = (x^1, \dots, x^n)$ (energies, masses, spatial and angular variables...) recorded in a brief span of time by physics detectors. The events can be classified in different *species*, which are generally denoted with *signals*, for the events of interest for their physics phenomena, and *backgrounds*, all that remains. Each variable x^j is distributed for the given species s with a probability distribution function (PDF) $\mathcal{P}_s^j(x^j; \hat{\theta}_s^j)$, where $\hat{\theta}_s^j$ are free (not constant) parameters of the PDF. If the variables are uncorrelated each other, then the total PDF for the species s is expressed by

$$\mathcal{P}_s(\hat{x}; \hat{\theta}_s) = \prod_j \mathcal{P}_s^j(x^j; \hat{\theta}_s^j). \tag{1}$$

The ML technique allows to estimate the values of the free parameters, as well the number of events belonging to each species n_s, by maximizing the function

$$\mathcal{L} = \frac{e^{-\sum_s n_s}}{N!} \prod_{i=1}^{N} \sum_s n_s \mathcal{P}_s(\hat{x}_i; \hat{\theta}_s), \tag{2}$$

which is called *extended likelihood function*. We should underline that $\hat{x}_i$ are measured and the $\mathcal{P}_s^j$ functions are well-known, so $\mathcal{L}$ only depends on the free parameters we want to fit on the data sample. The search of maximum for $\mathcal{L}$ can be carried out numerically. Usually, it is used to minimize the equivalent function $-\ln(\mathcal{L})$, the *negative log-likelihood* (*NLL*). So the *NLL* to be minimized has the form[1]:

$$NLL = \sum_s n_s - \sum_{i=1}^{N} \left(\ln \sum_s n_s \mathcal{P}_s(\hat{x}_i; \hat{\theta}_s) \right), \tag{3}$$

that is a sum of logarithms. The most common method used in the high energy physics (HEP) community for the minimization is based on the MIGRAD algorithm inside the MINUIT package. MIGRAD performs the minimization using the *variable metric* method [3]. This method involves the calculation of the derivatives of the *NLL* for each free parameter. Since very often we are deal with minimizing a function for which no derivatives are provided, MIGRAD is able to estimate the derivatives of the function by finite differences [4]. The whole procedure of minimization requires several evaluations of the *NLL*, which requires itself the calculation of the corresponding PDFs for each variable and each event of the data sample. Hence, depending on the complexity of the *NLL* function with several free parameters, many independent variables and large data samples, the minimization

[1] We omit the $N!$ term in the expression, which does not depend on the parameters.

procedure can be very time-consuming. In this case it is important (or even mandatory) to speed-up the evaluation of the *NLL* [5]. The common software used in HEP community for the evaluation of the *NLL* is RooFit [6], which is part of the general data analysis framework ROOT [7]. Currently RooFit implements an algorithm for the *NLL* evaluation which cannot take fully advantage from data vectorization and other code optimizations (like function inlining) due to its implementation based on C++ virtual methods [5]. To overcome these limitations, we have designed and implemented a new optimized algorithm on CPUs, and parallelized it using a data parallelism paradigm implemented with OpenMP. The algorithm has been also implemented to run on a Graphics Processing Unit (GPU) device by using CUDA language provided by NVIDIA. This work is thoroughly documented in [1].

1.2 NLL evaluation

RooFit package is formed by a set of C++ classes constructed on top of ROOT framework dedicated to likelihood-based analyses. Basically for each mathematical concept there is a corresponding C++ class, e.g. classes for the PDFs and variables definition. Then there is a special class which takes care of finalizing the *NLL* calculation. Furthermore RooFit provides an interface to the MINUIT package. We should underline that all floating point operations are performed in double precision. Data are organized in memory like a matrix where the columns contain the values for each variable, and the rows represent the values of the variables belonging to each event. All classes for PDFs inherit from a common abstract class, which provides the common interface. So each class has a virtual method to get the value of the PDF. Combinations of PDFs are possible with classes for adding, multiplying and convoluting basic PDFs. In order to calculate the *NLL* from the formula (3), the current available RooFit algorithm consists of the following steps (in order):

- For a given set of values of *NLL* free parameters, loop over the events $i = 1...N$:
  - read the values of the variables for event i;
  - calculate the PDFs for the event i;
  - combine, by means of addition and multiplication, the results of the individual PDFs to calculate the total PDF value for the event i;
  - calculate the logarithm of the total PDF value, which is the term in the sum of the *NLL*;
  - accumulate the terms of the sum.
- Finalize the calculation of the *NLL*.

The key part of this procedure is the calculation of *all* PDFs for *each* event, and then there is a single loop over all events. Since this is done by having recourse to calls of the virtual method of each PDF, this algorithm does not allow particular code optimization, like inlining and data vectorization, and it introduces the obvious overhead due to the virtual method calls. In order to take benefit from code optimization, we redesigned the algorithm to reduce the number of calls to virtual methods. Furthermore, the data are stored differently: the values of each variable are organized in independent arrays, so that we can profit from the coalescing of memory accesses for each variable. The new algorithm follows a different procedure with respect to the RooFit algorithm described above:

- For a given set of values of the parameters and a given PDF, we evaluate the PDF on each event of the data sample (which means calculating the PDF on the corresponding arrays of variables), and we save the results of this calculation in an array. So we do a loop over all events $i = 1...N$ and calculate the PDF for each of them.

- Repeat the previous step for all PDFs, so we end up with several arrays of partial results (an array for each PDF). Each array of results is composed by N elements, i.e. a result for each event.
- Combine, by means of addition and multiplication, all arrays of partial results, corresponding to each event, providing a final array of results, i.e. the array of results of the total PDF.
- Calculate the logarithm of the total PDF results.
- Do the sum of the total PDF results and finalize the calculation of the *NLL*.

The key part of this procedure is the calculation of *each* PDF for *all* events, so that instead of one single *global* loop over the events, now we have independent *local* loops for each PDF (and their combinations). For the implementation of this new algorithm in RooFit, we add a new virtual method for each PDF class with a reference to the data sample as parameter. Inside this method we perform the local loop over all values of the variables of the corresponding PDF, storing the results of the calculations in an array of partial results. Then the method returns a reference to this array. Since this new virtual method is called just once per each PDF during a *NLL* evaluation, and then within local loops we perform the calculations of the mathematical functions for all events, we can conclude that the number of calls to virtual methods does not depend by the number of events. Furthermore, thanks to the new data structure organized as arrays for each variable, this code can easily be vectorized for the calculation of each PDF. The loop over the final results of the total PDF to calculate and finalizing the *NLL* evaluation is done in the usual class for the *NLL* finalization. We should note that a drawback of this new algorithm is that we have to manage all the arrays for the temporary results.

2 *NLL* Evaluation with OpenCL

The NLL evaluation requires an evaluation of a function over a vector of elements which can be easily parallelized using OpenMP. An example of an evaluation of a Gaussian function is shown in figure 1a. This code is auto-vectorized if compiled by the Intel compiler, so the programmer don't need to think about that. Also, work partitioning is automatically taken care of by OpenMP.

```
void evaluatePdfGaussian(const double mu, const double sigma, const double* data,
  double* results, const int N)
{
  #pragma omp parallel for
  for(int i = 0; i < N; i++)
  {
    double temp = (data[i]-mu)/sigma;
    temp *= temp;
    results[i] = exp(-0.5*temp);
  }
}
```

(a) OpenMP

```
 __kernel void evaluatePdfGaussian(__const double mu, __const double sigma, __global
  const double *data, __global double *results, __const int N)
{
  int i = get_global_id(0);
  if (i >= N) return;
  double x = data[i];
  double temp = (x-mu)/sigma;
  temp *= temp;
  results[i] = exp(-0.5*temp);
}
```

(b) OpenCL

Figure 1: OpenMP and OpenCL versions of a Gaussian evaluation function. Mu and sigma are doubles and data and results are pointers to arrays of double.

AMD has recently released an OpenCL SDK called AMD App SDK (v. 2.3 released on 29.01.2011), which supports programming both AMD GPUs and x86-compatible CPUs. In this work we therefore use the AMD App SDK for the CPU and the NVIDIA OpenCL SDK for the GPU. Being able to program any device in the same programming model seems ideal, but unfortunately this does not come without a penalty. We will in the next sections describe our experiences with targeting different devices in the same program and with the same programming model, and try to highlight the limitations that we find most critical. The goal would be to be able to use the OpenCL kernel in figure 1b for both CPUs and GPUs, and that it would be comparable performance-wise to the OpenMP version.

3 Drawbacks of OpenCL on CPUs

Threads on modern GPUs are very lightweight, and scheduled by hardware mechanisms. In an application performing calculations on e.g. vectors, it is therefore appropriate to make each thread typically target one single element in that vector. If the number of threads are smaller than the number of

```
__kernel __attribute__((vec_type_hint(double2))) void evaluatePdfGaussian(__const
    double mu, __const double sigma, __global const double *data, __global double *
    results, __const int N)
{
  int i = get_global_id(0);
  if (i >= N/2) return;
  double2 x = vload2(i, data);
  double2 temp = (x-mu)/sigma;
  temp *= temp;
  double2 result = exp(-0.5*temp);
  vstore2(result, i, results);
}
```

Figure 2: The vectorized version of the kernel in figure 1b

elements, threads that are done with their execution will be rescheduled to compute another element without much scheduling overhead. This is automatically taken care of by the OpenCL library/driver, which clearly eases the programming effort. It is tempting to use one unified programming model for a range of devices, however, using the OpenCL implementation in AMD App SDK for the CPU is not necessarily straightforward if you want to achieve performant code. Figure 1b shows an OpenCL kernel suitable for running on a GPU. Using this implementation for the GPU we achieve nearly equal performance as the corresponding CUDA implementation. It would be ideal if this kernel could be highly performant also on the CPU. However, this is not the case for two main reasons: AMD App SDK does no auto-vectorization and the way to do threading on the GPU is different from the CPU.

3.1 Explicit vectorization

Since the OpenCL compiler in the AMD App SDK does not support autovectorization, one has to explicitly program with vector types defined in the OpenCL standard to make the compiler produce vectorized x86 code (SSE). By introducing explicit vectorization in the code we now have a different kernel, and the benefit of a unified programming model is dramatically reduced, since we have to have one edition of the same kernel for both GPUs and CPUs[2]. It would be a lot easier if the OpenCL compiler had autovectorization capabilities. Figure 2 shows the vectorized edition of the kernel in figure 1b. On the other hand, the syntax for loading and storing vector types in OpenCL is clean and easy to understand, and vector types have mathematical operators implemented.

Note that since we have to program this by hand, we have to have arrays of even length the way the source code is now. Also, if we want to use larger vector registers in the future (e.g. AVX), we

[2]AMD/ATI GPUs could also benefit performance-wise when using vector data types, since it may make it easier for the compiler to pack data into the VLIW units of the AMD cards. This is not the case with NVIDIA GPUs, which in practice would give us one kernel for each GPU vendor too. Note that this is not a necessity to achieve good performance, but it's still there.

```
__kernel __attribute__((vec_type_hint(double2))) void evaluatePdfGaussian(__const
    double mu, __const double sigma, __global const double *data, __global double *
    results, __const int N, __const int numComputeElements)
{
  int i = get_global_id(0);
  if (i >= N) return;
  int part = N/numComputeElements;
  for(int index = i*part; index < (i+1)*part - 1; index+=2)
  {
    double2 x = vload2(index/2, data);
    double2 temp = (x-mu)/sigma;
    temp *= temp;
    double2 result = exp(-0.5*temp);
    vstore2(result, index/2, results);
  }
}
```

Figure 3: Same as figure 2, but with more work per thread

have to change all the vector types in the code, which is painfully inflexible (it could maybe be possible with a typedef workaround, though).

3.2 Thread scheduling

With the AMD App SDK, the kernel in figure 2 will not perform very well on a CPU, even though vectorized code is emitted. This has to do with how the AMD App SDK handles thread scheduling internally. We don't know the details under the hood, but AMD encourage users (in their online examples) to give CPU threads more work than GPU threads. Tests we've conducted, showed that running CPU kernels as the one in figure 2 resulted in a performance achievement of roughly 33% compared to an autovectorized OpenMP version. However, by splitting the data into an appropriate amount for each thread, we achieved speeds similar to it.

Figure 3 shows a kernel that does more work per thread. Note that this kernel assumes that the number of compute elements divides N. Doing more work per thread forces the developer to think about work distribution, and then most of the benefits with this programming model are lost. This is not very difficult to achieve, but it should be implicit in the programming model, just as it is when targeting GPUs. It does not make any sense to use different OpenCL implementations tuned for each device, since then you're basically back to where you started. The ideal case would be to have the same kernel for both the CPU and the GPU ("SIMT for the CPU"), and that the OpenCL SDK took care of this automatically. As we clearly see, the kernels in figure 1b and 3 respectively are very different from each other. Also, note that the Gaussian function is a trivial function to implement. Other functions (or kernels in general) might be much more complex.

4 Conclusion

The way the AMD App SDK implements the OpenCL standard is inappropriate for multicore applications that are written to be performant and easily programmable/modifiable. The concept of lightweight threads is reflected strongly in the programming model, but the AMD App SDK does not implement this effectively for CPU threads. An ideal implementation should make it possible to write kernels in a "one element per thread" style for CPUs and deliver high performance. Leaving this work for the programmer leads to an uneffective framework to develop multicore programs in.

In addition, we think autovectorization is necessary for making OpenCL attractive for multicore programming. It is always a plus to have vector types to customarily optimize programs, but being forced to implement vectorization on a source level basis across entire OpenCL programs is tedious.

All in all, having to do these two major changes to the code doesn't make our application easily extendable. It may be easy for someone on the outside suggesting major features such as autovectorization and work-distribution, without regarding the amount of work required to implement it. However, the vendors providing OpenCL implementations does exactly this in their existing products, and one can never escape the fact that by the way the AMD App SDK is implemented for multicore CPUs today, it is inferior to other multicore programming tools such as OpenMP and TBB that can lean themselves on auto-vectorizing compilers, while still delivering a relatively pleasant interface for programmers.

It is worth to mention that there is another project by the PGI group which aims to run CUDA C code on x86 CPUs [8] with good performance (e.g. supporting SIMD units). Ideally, an OpenCL SDK would also support this.

5 Intel OpenCL wish list

Since we do most of our work in Linux (as many others), a Linux version of OpenCL (maybe an alpha version) from Intel in near future would be welcome. We also hope that Intel would consider looking into the two main problematic areas we've highlighted in this report; autovectorization and making the work distribution among threads implicit in the programming model, since our final goal is to have a common implementation for CPUs and GPUs. As long as these two areas aren't implicit, using OpenCL as a solution for both CPUs and GPUs is inappropriate for the work we are doing.

References

[1] S Jarp, A Lazzaro, J Leduc, A Nowak, and F Pantaleo. Parallelization of maximum likelihood fits with openmp and cuda. http://cdsweb.cern.ch/record/1328927, 2011.

[2] G. Cowan. *Statistical Data Analysis*. Clarendon Press, Oxford, 1998.

[3] W. C. Davidon. Variable metric method for minimization. *SIAM J. Optim.*, 1:1–17, 1991.

[4] F. James. In *MINUIT - Function Minimization and Error Analysis*, CERN Program Library Long Writeup D506, 1972.

[5] A. Lazzaro and L. Moneta. *J. Phys.: Conf. Series*, 219:042044, 2010.

[6] W. Verkerke and D. Kirkby. The roofit toolkit for data modeling. proceedings of PHYSTAT05. Imperial College Press, London, 2006.

[7] R. Brun and F. Rademakers. Root - an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A*, 389:81, 1997.

[8] http://www.pgroup.com/resources/cuda-x86.htm. Accessed 24.03.2011.

Page intentionally left blank