Implementing Data Parallelisation in a Nested-Sampling Monte Carlo Algorithm

Wim Vanderbauwhede
School of Computing Science
University of Glasgow
Glasgow, UK

Stefanie Lewis, David Ireland
School of Physics & Astronomy
University of Glasgow
Glasgow, UK

***Index Terms*—General-Purpose computation on Graphics Processing Units (GPGPU), Parallelization of Simulation**

***Abstract*—In this paper we report our work on the parallelisation of a Nested Sampling Monte Carlo algorithm used in the nuclear physics field of hadron spectroscopy. The purpose of the application is to fit a set of parameters in a nuclear physics model based on the observations of the beam properties.**

We used both OpenCL and OpenMP to parallelise the existing code. Our aims where to achieve parallelisation with minimal changes to the original source code, and to evaluate the performance of the parallel code on both a GPU and a multicore CPU.

On the implementation side, we show that by using our OclWrapper abstraction over the OpenCL API, integration of OpenCL code into and existing C++ code base is much simplified, to the extent that integrating OpenCL is not considerably more effort than using OpenMP, as the main effort is in making the code suitable for parallel execution.

Our evaluation shows that the best results depend strongly on the size of dataset. For large numbers of events (10^5), we achieved a best speed-up of $22\times$ using OpenCL on the CPU. For small numbers of events (10^3), we achieved a best speed-up of $4\times$ using OpenMP on the CPU. The best GPU speed-up was $7\times$ for 10^5 events. This is mainly a result of the data transfer time, which more than offsets the improvement in computation time.

I. Background

A. Hadron Spectroscopy

In theoretical physics, quantum chromodynamics (QCD) is a theory of the interactions between quarks and gluons which make up hadrons (such as the proton, neutron or kaon). Quantum chromodynamics does well to explain the strong interaction, but ultimately fails at the mass level of a proton. For this, we look at quark models, such as the diquark models or the symmetric quark model. These models each predict a set of *resonances* ((unstable particles), and if we can experimentally find resonances predicted by some quark models and not others, we can learn valuable information about the nucleons (protons and neutron).

One way of finding these resonances is through *pseudoscalar meson photoproduction*, a physical process which can be completely described by four complex amplitudes. In pseudoscalar meson photoproduction, an incoming photon beam is incident on a stationary nucleon target. Several scattering products (and decay products) are then emitted and detected by a large acceptance detector. These amplitudes are experimentally accessible through 16 polarisation observables, which are all correlated through these amplitudes. Through different experimental setups (such as transverse target polarisation, linear beam polarisation, etc) we can access different observables.

Using nested sampling analysis, we explore the amplitude space rather than treating observables as independent free variables. By doing so, we maximise the information we can extract from the experimental data and all correlations are inherently kept in tact. From recent results, it is clear that this approach gives us much more information about polarisation observables to which the particular experimental setup is not sensitive.

B. Nested Sampling Monte Carlo

Nested sampling [1] is a form of Markov Chain Monte Carlo, a Bayesian statistics approach to data analysis. Whilst nested sampling was applied to a specific nuclear physics problem in this paper, it is, at its core, a general algorithm that could be applied to a wide range of problems across all disciplines.

The primary objective of nested sampling is to provide a future-proof model comparison tool. An initial distribution, called the prior, is used in conjunction with an event-by-event likelihood function to generate a final distribution (known as the posterior) and a value commonly referred to as evidence, Z. This evidence value is the feature that allows for straightforward, time-independent comparisons of model assumptions.

For this work the focus is on the output of a posterior distribution (rather than the evidence). A prior consisting of points distributed across a physically constrained region of space is generated. An event-by-event likelihood function is applied to this prior, and the resulting posterior distribution is examined.

For each point in the prior, a likelihood value is calculated based on a problem-specific likelihood function. The point with the lowest likelihood is recorded and overwritten with a copy of a surviving point. This new point is then altered and its likelihood is calculated. If the resulting likelihood is lower than that of the overwritten point, the new point is moved again. This process continues until the likelihood of the new point is greater than that of the overwritten point. The algorithm then finds the next point with the lowest likelihood and the process

is repeated until a given termination condition is met [2]. For consistency, the termination condition used in this work was a set number of iterations.

C. Data Parallelisation

Since the clock speed of CPUs has stagnated, parallel programming has become the focus of computing performance development. The use of multicore processors and General-Purpose Graphics Processing Units (GPGPUs) has become mainstream in everything from scientific computing and state-of-the-art gaming technology to standard desktop computers and laptops. There is now a sustainable path to improving computing technologies for the foreseeable future. Although the spotlight is currently on GPGPU computing, it must be remembered that all programs and algorithms will include some amount of sequential code, even if it exists solely to execute kernel functions or perform or some standard initialisations. In most cases, these serial sections of a program create bottlenecks that no amount of parallelisation can avoid. For this reason, heterogeneous platforms – i.e. the combination of highly optimised CPU cores with the massively parallelisable GPU cores – have become increasingly popular. These two components must complement each other – if the CPU is outdated and obsolete, any speed-up obtained from a high-end GPU will be hidden by the slow processing at one of these bottlenecks. In order to make the most of the available hardware, both components must be taken into consideration.

Although initially used to categorise the various types of computer hardware systems, Flynn's taxonomy has recently been applied to software and algorithms: Single Instruction stream Single Data (SISD) refers to a sequential algorithm (one piece of data being used in a single operation), Single Instruction stream Multiple Data (SIMD) includes programs that perform the same operation on many data concurrently. SIMD algorithms are at the heart of data parallelisation. There are many cases in data analysis where an algorithm performs operations on many data points independently. For these situations, a considerable speed-up can be achieved by dividing the data set into smaller sections and performing the same operations in parallel, with one thread handling one section of data.

Not all algorithms can be parallelised; recursive and sequential programs, or even serial sections of code, can form bottlenecks that impede the run-time of a program. There are some cases where parallelising data over multiple threads or cores can result in a slower run-time as no speed-up is gained and time is lost during data transfer or thread initialisations. Even in cases where an algorithm lends itself naturally to parallelism, it is crucial to understand exactly where and how it should be done in order to obtain the greatest benefit.

D. CPU/GPU System Performance

It is important to have an idea what to expect in terms of performance of a GPU-accelerated application. The achievable speed-up from offloading an application to a GPU can be analysed as a function of the computational speed up (which in its term depends on the hardware parallelism, clock speed and memory bandwidth) and the data transfer speed. Figure 1 shows a generic graph which can be used to assess the performance of an algorithm.

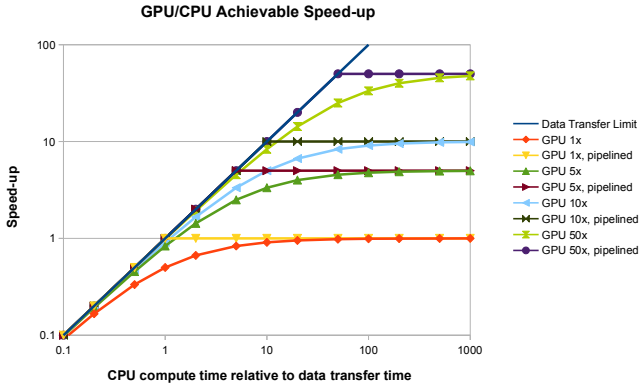

Fig. 1. Achievable speed-up from offloading work to the GPU

What the graph shows is the achievable speed-up as a function of the CPU compute time relative to the data transfer time, with the GPU/CPU computational speed-up as a parameter. For example, if the computation on the CPU takes 100ms, and the data transfer 1000ms, then there can be no speed-up, no matter how fast the GPU computes. On the other hand, if the CPU takes 1000ms and the transfer time is 100ms, then with a GPU/CPU computational speed-up of 5× the total speed-up = 1000 / (100+1000/5) = 3.3x. The curved lines assume overlapping of data transfers and computation ("pipelined" in the legend); the straight lines assume alternating data transfers and computations. It is clear that overlapping only makes a significant difference if the data transfer and computation times are of the same order.

To facilitate the analysis, we define the computation performance indicator as *CPI= #threads* × *SIMD vector size* × *clock speed*. This number is directly proportional to flops, but more easy to obtain from datasheets. We define "threads" as the product of the number of cores and their hyperthreading capability; "SIMD vector size" is defined as the number of single-precision floating-point operations that can be performed in parallel, so on a CPU this is e.g. the width of the SSE or AVX vector instructions; on a GPU it would be the number of processing elements per compute unit. In Section II-C we will use the CPI to assess the expected performance of the hardware platform used in this work.

II. Methodology

The inputs for the Nested Sampling algorithm are a set of observations from a hadron beam scattering experiment. When the scattered products are detected, their energies and scattering angles are found. In our experiment, ϕ is the scattering angle of the kaon particle. The beam was linearly polarised and a variable *pol* indicates whether the beam was parallel or perpendicular for a given event. The only observable used

in the (simplified) likelihood computation is the photon beam asymmetry B. P_γ refers to the energy of the beam. For more details on the physics experiment we refer the reader to [3].

The original code for the Nested Sampling analysis was written in C++ using CERN's ROOT toolkit [4]. Analysis of the algorithm and run-time profiling show that the likelihood calculation is the most computationally intensive step and that it can be parallelised. The calculation can be expressed as

$$\tilde{A} = \frac{P_\gamma.B.cos(2\phi_i) + \delta_L)}{1 + \delta_L.P_\gamma.B.cos(2\phi_i))}$$

$$P_i(pol_i, \phi_i) = \begin{cases} pol_i = 0: & \frac{1}{2(1+\tilde{A})} \\ pol_i = 1: & \frac{1}{2(1-\tilde{A})} \end{cases}$$

$$L = \sum_{i=1}^{N} log(P_i(pol_i, \phi_i))$$

To parallelise this calculation, partial sums for $log(P_i)$ are computed in each thread, and a final accumulation over the number of threads is performed.

We implemented the parallel code using OpenCL (GPU/CPU) and OpenMP (CPU only).

A. OpenCL Integration

OpenCL [5] was developed by the Khronos Group in 2008 as an open standard for parallel programming of heterogeneous systems. It provides an API for control and data transfer between the host and device (typically the host CPU and a GPU) and a language for kernel development. Contrary to proprietary solutions such as Nvidia's CUDA and Microsoft's DirectX, OpenCL is open and cross-platform, so that it can be deployed on different operating systems (Linux, OS X, Windows) and hardware architectures (multicore CPUs, GPUs, FPGAs). The OpenCL API is defined for C and a C++. In practice, the API is quite fine grained and verbose and requires a lot of boiler plate code to be written. Consequently, it is not straightforward to integrate OpenCL in existing codes, especially for non-computing scientists. To facilitate the integration of the OpenCL code into the existing code base, we developed the OclWrapper library[1] which supports C, C++ and Fortran-95. The library wraps the OpenCL platform, context and command queue into a single object, with a much smaller number of calls required to run an OpenCL computation. As it is a thin wrapper, the additional abstraction comes at no cost in terms of features: the OpenCL API is completely accessible. The main API methods and their usage are illustrated in Algorithm 1 (*Hello, World*)

B. OpenCL versus OpenMP

Apart from the OpenCL implementation, and in order to assess the performance, we also implemented the kernel in OpenMP [6], an API for parallel programming on shared-memory multicore platforms. The OpenMP API consists of a set of preprocessor directives (pragmas) and function calls.

[1]https://github.com/wimvanderbauwhede/OpenCLIntegration

Algorithm 1 Hello, World in OpenCL using the C++ OclWrapper API

```
#include "OclWrapper.h"
int main () {
    const int strSz=17;
    char cc1='\n';
    char cc2=' ';
  // Instantiate the OpenCL Wrapper
    OclWrapper ocl();
  // Load the Kernel from file
    ocl.loadKernel(
        "helloworld.cl","hello_world");
    ocl.createQueue();
  // Create Buffers for reading & writing
    cl::Buffer& str_buf=
        ocl.makeWriteBuffer(strSz);
    cl::Buffer& c1=ocl.makeReadBuffer(1);
    cl::Buffer& c2=ocl.makeReadBuffer(1);
     ocl.writeBuffer(c1,1,&cc1);
    ocl.writeBuffer(c2,1,&cc2);
  // Enqueue the Kernel, NullRange by default
    ocl.enqueueNDRange();
  // Run the Kernel and
  // wait for it to finish
    ocl.run(str_buf,c1,c2).wait();
  // Read back the result
    char* str=(char*)malloc(strSz);
    ocl.readBuffer(str_buf,strSz,str);
  // Display the result
    std::cout <<str;
  // Exit
    return 1;
}
```

The OpenCL kernel and the corresponding OpenMP code are shown in Algorithms 2 and 3. The differences between both are quite small, and fall into three categories: first, the OpenMP code is object-oriented C++ code, so some of the variables (e.g. angles, nEvents) are class attributes and not subroutine arguments. In OpenCL all input and output variables must be kernel arguments in the `__global` memory space and the subroutine must be `__kernel void`, so the kernel signature is longer. Second, OpenMP requires pragmas (`#pragma omp ...`) to control the parallelism and indicate which variables are shared. Finally, the most important difference is that OpenMP parallelises the complete loop implicitly, whereas each instance of the OpenCL kernel executes explicitly on part of the loop. This requires the use of the OpenCL API functions such as `get_global_id()` etc.

In the OpenMP code, we compute the partial sums per OpenMP thread and aggregate them. In our OpenCL code, the loop is divided into parts to be executed in every thread of every compute unit using the global id; we create a partial sum in every thread and aggregate it per compute unit. To ensure that the computation of all partial results in the workgroup was finished, we insert the barrier() call. Finally, we return the result for each compute unit via the array `LogL[group_id]`. The sum over all compute units is performed on the host. This partitioning between the threads and compute units is obtained by setting the NDRange:

Platform	#Cores	Vector size	Clock speed (GHz)	CPI	Memory BW (GB/s)
Intel i7-2700K	12	8	3.5	336	21
Nvidia Tesla S2075	14	32	1.15	515.2	144

TABLE I
SPECIFICATIONS OF HARDWARE PLATFORMS USED IN THIS WORK

```
ocl.enqueueNDRange(
    cl::NDRange(compute_units*NTH),
    cl::NDRange(NTH));
```

The global work range is `compute_units*NTH`, i.e. the total number of threads on the GPU; the local range is `NTH`, the number of threads per compute unit.

C. Hardware Platform

The host CPU used in this work is an Intel i7-2700K running at 3.5GHz. It is a quad-core CPU with hyperthreading. The maximum memory bandwidth is 21 GB/s. This processor has 256-bit AVX SIMD, so a smart compiler will do up to 8 floating point operations in parallel. The GPU used is an NVidia Tesla S2075 GPU with 14 compute units (448 "cores") running at 1.15 GHz. The maximum memory bandwidth is 144 GB/s. The host-device connection is a 16-lane Gen2 PCIe bus. The CPIs are show in Table I

We observe that purely in terms of computation, under optimal circumstances, the Tesla GPU can be at best 1.5× faster than the Intel CPU. If the memory bandwidth would be the limiting factor (as opposed to computation), the achievable speed-up for the application running on the GPU would be 6.8×. The total achievable speed-up is limited by the data transfer rate between host memory and GPU memory, and the overhead for control of the GPU. In Section III-C we present the detailed discussion of the cost of data transfer and computation. However, as our application is not memory bandwidth limited, the CPI shows that even ignoring the transfer time, the speed-up from the GPU will be very modest, and that taking into account the transfer time, we expect the GPU to perform less well than the CPU.

III. Results and Discussion

In our design of experiments we varied the number of events $(10^3, 10^4, 10^5)$ and the number of threads (2 to 64), and the number of iterations (from 10K to 150K). Each experiment was repeated 20 times, we verified that variation was negligible and computed the averages.

A. Optimal Number of Threads

We performed tests to determine the optimal number of threads using each dataset on the three implementations that allowed for multithreading. The number of threads over which to parallelise the data was changed and the program runtime was measured.

The Figures 2, 3 and 4 illustrate that the optimal number of threads is dependent on the size of the dataset, and differs for the various implementations. The large variation in the OpenMP performance is due to the fact that the OpenMP performance improves for growing numbers of threads as long as there are fewer OpenMP threads than physical threads. Once the number of OpenMP threads exceeds the number of physical threads, the performance deteriorates strongly. This effect is also observable for OpenCL on the CPU, but much less pronounced, because OpenCL schedules the threads so that they do not compete with one another.

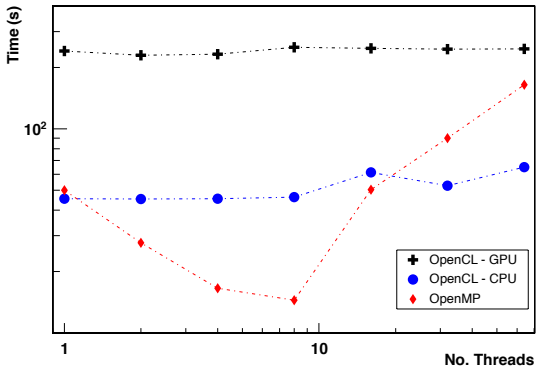

Fig. 2. Thread test results from dataset with 1000 events.

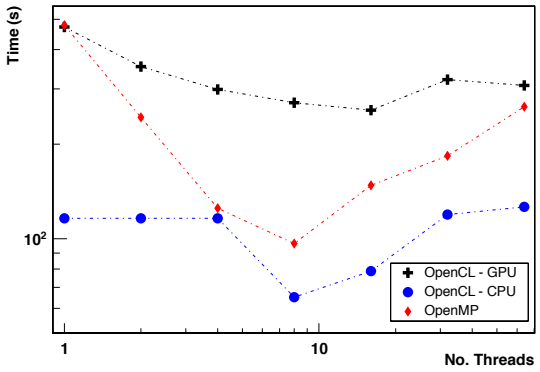

Fig. 3. Thread test results from dataset with 10000 events.

B. Optimal Performance

Figure 5 shows the performance for the optimal number of threads for each implementation, as a function of the number of events. As expected, we observed no dependency on the number of iterations (50K in the Figure). The most interesting observation is the difference in behaviour between the OpenCL and OpenMP parallel code: the execution time of the OpenMP code (for 8 threads) is proportional to number of events; however, the OpenCL code performs significantly better for large numbers of events. Interestingly, the OpenCL code results in a larger speed-up than suggested by the number of hardware threads on the CPU, which indicates that the Intel OpenCL compiler also uses the vector support of the CPU. As

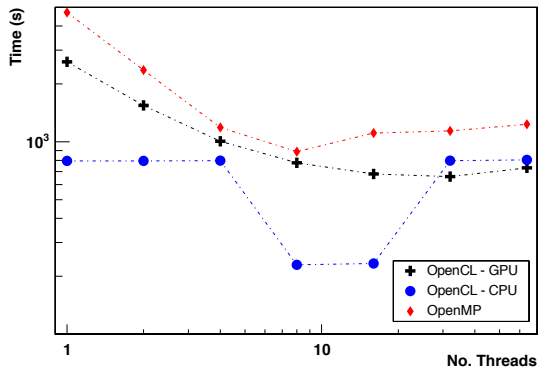

Fig. 4. Thread test results from dataset with 100 000 events.

a result of this different behaviour, there is no single optimal implementation:

For large numbers of events (10^5), the best choice is OpenCL on the CPU, with a speed-up of up to 22×. For small numbers of events (10^3), the best choice is OpenMP, the achievable speed-up is smaller, up to 4× For intermediate numbers (10^4), best speed-up was 8× using OpenCL on the CPU. The best GPU speed-up was 7× for 10^5 events. This is mainly a result of the data transfer time, which more than offsets the improvement in computation time.

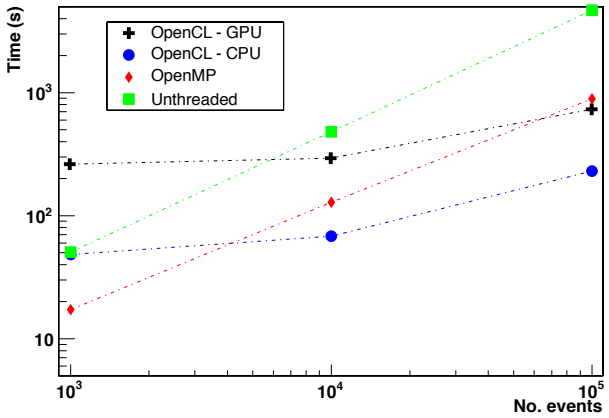

Fig. 5. Performance for optimal number of threads

C. Impact of Data Transfer Time

The amount of time spent merely transferring data to a given compute device using OpenCL was measured by running the program with an empty likelihood kernel function.

The data transfer time is linear with the dataset size but has a significant constant offset. This offset is the time taken to control the GPU, i.e. not the actual transfer time. The corresponding time for OpenCL on the CPU is about 10× smaller. The dataset sizes used in this work are typical for problems in hadron spectroscopy but are fairly small in absolute size.

TABLE II
DATA TRANSFER RESULTS PER ITERATION FOR OPENCL ON THE GPU.

#Events	Data transfer time (ms)	Total time (ms)	% Data transfer
1000	0.227 ± 0.008	0.277 ± 0.0023	82
5000	0.231 ± 0.010	0.328 ± 0.0027	70
10000	0.232 ± 0.004	0.3997 ± 0.0041	58
100000	0.229 ± 0.005	1.560 ± 0.0010	15

TABLE III
DATA TRANSFER RESULTS PER ITERATION FOR OPENCL ON THE CPU.

#Events	Data transfer time (ms)	Total time (ms)	% Data transfer
1000	0.021 ± 0.0007	0.0457 ± 0.001	46
5000	0.021 ± 0.0022	0.0776 ± 0.001	27
10000	0.021 ± 0.0007	0.114 ± 0.0006	18
100000	0.021 ± 0.0011	0.790 ± 0.0074	3

These results illustrate that a significant portion of the runtime for the GPU is spent handling the control of the device. In algorithms such as nested sampling, where the kernel is invoked many times for relatively small calculations, data parallelism on the GPU is not necessarily the most effective solution. Parallelising data and running on the CPU provides a much more noticeable speed-up, partly due to this overhead. This is in line with our CPI-based analysis.

CONCLUSION

Our work demonstrates that OpenCL can be used successfully to accelerate a Nested-Sampling Monte Carlo Algorithm. Thanks to our novel OclWrapper library, the OpenCL integration requires only a small additional effort compared to OpenMP. Our work also shows that the best choice of implementation and hardware platform depends very much on the size of the events data set. In any case, for this particular algorithm, the amount of computations is too small to outweigh the cost of the data transfer to the GPU, so a multicore CPU is a better choice. For large numbers of events (10^5), we achieved a speed-up of up to 22×. In future work we want to investigate if we can convert more portions of the Nested Sampling algorithm into OpenCL kernels, in order to reduce the data transfer between host and device.

REFERENCES

[1] J. Skilling, "Nested sampling for general bayesian computation," *Bayesian Analysis*, vol. 1, no. 4, pp. 833–859, 2006.
[2] D. Sivia and J. Skilling, *Data Analysis: A Bayesian Tutorial*, ser. Oxford Science Publications. OUP Oxford, 2006. [Online]. Available: http://books.google.de/books?id=zN-yliq6eZ4C
[3] D. Ireland, "Information content of polarization measurements," *Physical Review C*, vol. 82, no. 2, p. 025204, 2010.
[4] R. Brun and F. Rademakers, "ROOT—an object oriented data analysis framework," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1, pp. 81–86, 1997.
[5] A. Munshi *et al.*, "The opencl specification," *Khronos OpenCL Working Group*, vol. 1, pp. 11–15, 2009.
[6] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

Algorithm 2 OpenCL kernel

```
__kernel void Likelihood(__global float* B_Pg,
__global float* angles, __global float* pols,
__global float* LogL, const float Log2e,
const int Asize, const int nunits) {

    int group_id=get_group_id(0);
    int glob_id = get_global_id(0);
    int th_id = get_local_id(0);

    float B=B_Pg[0];
    float Pg=B_Pg[1];

    float  delta_L      = 0.0;
    float  LogL_2       = 0.0;
    float  LogL_2_th      = 0.0;
    local  float LogL_2_array[NTH];
    int mSz = Asize/(nunits*NTH);
    int start = glob_id*mSz;
    int stop = start+mSz
    for (int idx = start; idx<stop; idx++) {
        float angle = angles[idx];
        float pol   = pols[idx];
        float costerm = Pg*B*cos(2*angle);
        float A_tilde = (costerm + delta_L)
                    /(1+(costerm*delta_L));
        float prob  = 0.0;
        if (pol < 0){
            prob = 0.5*(1 + A_tilde);
        } else {
            prob = 0.5*(1 - A_tilde);
        }
        LogL_2_th += log2(prob);
    }
    LogL_2_array[th_id] = LogL_2_th;
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int i = 0; i < NTH; i++){
        LogL_2 += LogL_2_array[i];
    }
    LogL[group_id] = LogL_2 / Log2e;
}
```

Algorithm 3 OpenMP kernel

```
#include <omp.h>

class NS {
    public:
    // ...
    virtual float Likelihood (float B, float Pg);
// ...
    float *angles;
    float *pols;
    int nEvents;
    // ...
}

float NS::Likelihood(float B, float Pg) {
    float delta_L  = 0.0;
    float LogL     = 0.0;
    float *LogL_2_array = new float[NTH];
    for (int i = 0; i < NTH; i++){
        LogL_2_array[i] = 0.0;
    }
    float LogL_2   = 0.0;
#pragma omp parallel private(delta_L) \\
    shared(LogL_2_array) num_threads(NTH)
    {
        int th_id = omp_get_thread_num();
        float LogL_2_th=0.0;
#pragma omp for
        for (int idx = 0; idx < nEvents; idx++){
            float angle = angles[idx];
            float pol   = pols[idx];
            float costerm = Pg*B*cos(2*angle);
            float A_tilde = (costerm + delta_L)
                        /(1+(costerm*delta_L));
            float prob = 0.0;
            if ( pol < 0 ){
                prob = 0.5*(1 + A_tilde);
            } else {
                prob = 0.5*(1 - A_tilde);
            }
            LogL_2_th += log2(prob);
        }
        LogL_2_array[th_id] = LogL_2_th;
    } // omp parallel
    for(int i = 0; i < NTH; i++){
        LogL_2 += LogL_2_array[i];
    }
    LogL = LogL_2 / Log2e;
    return LogL;
}
```