Evaluation of likelihood functions on CPU and GPU devices

Sverre Jarp, Alfio Lazzaro, Julien Leduc,
Andrzej Nowak, Yngve Sneen Lindal
European Organization for Nuclear Research (CERN), Geneva, Switzerland

14th International Workshop on Advanced Computing and
Analysis Techniques in Physics Research,
Uxbridge, London, UK
September 5th–9th, 2011

Introduction/motivation

- Methods based on likelihood functions are used in fitting procedures to determine whether results from HEP experiments show promises of what is expected or not (e.g. in the RooFit package)
- In essence this means to fit a set of statistical parameters to a set of observed data from an experiment
- Fitting can be computationally complex and often involves computation of transcendental functions
- As accelerators become more complex and higher luminosities are reached, the amount of collected physics events grows
- This implies that large computational resources must be used. We therefore want to utilize parallelism in CPUs as effectively as possible, in addition to naturally parallel co-processors such as GPUs
- Our work is based on a RooFit prototype called *MLFit*

Likelihood-based Techniques

- **Data are a collection of independent events**
  - an event consists of the measurement of a set of variables/observables(energies, masses, spatial and angular variables…) recorded in a brief span of time by the physics detectors

- **Introducing the concept of probability** P **(= Probability Density Function, PDF) for a given event to be signal or background, we can combine this information for all events in the *likelihood function***

$$\mathcal{L} = \prod_{i=1}^{N} \mathcal{P}(\hat{x}_i|\hat{\theta})$$

N number of events
$\hat{x}_i$ set of observables for the event i
$\hat{\theta}$ set of parameters

- **Several data analysis techniques requires the evaluation of** L **to discriminate signal versus background events**
- **Finding the maximum of this function is equivalent to "what is the parameter estimation that makes the data set most probable for the prediction model?"**

Maximum Likelihood Fits

- **It allows to estimate free parameters over a data sample, by minimizing the corresponding Negative Log-Likelihood (**NLL**) function (extended likelihood)**

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

s species, i.e. signals and backgrounds
n_j number of events belonging to the species j

- **The procedure of minimization can require several evaluations of the** NLL
  - Depending on the complexity of the function, the number of observables, the number of free parameters, and the number of events, the entire procedure can require long execution time
  - Mandatory to speed-up the evaluation of the *NLL*

The model

$$n_a G_1^a(x) G_2^a(y) G_3^a(z) + n_b G_4^b(x) G_5^b(y) G_6^b(z) +$$
$$n_c A_1^c(x) P_1^c(y) P_2^c(z) + n_d P_3^d(x) P_4^d(y) A_2^d(z) +$$
$$n_e P_5^e(x) G_7^e(y) A_3^e(z)$$

Model from B. Aubert *et al.*,
Phys. Rev. D80, 112002, 2009

21 PDFs in total, 3 observables, 5 species

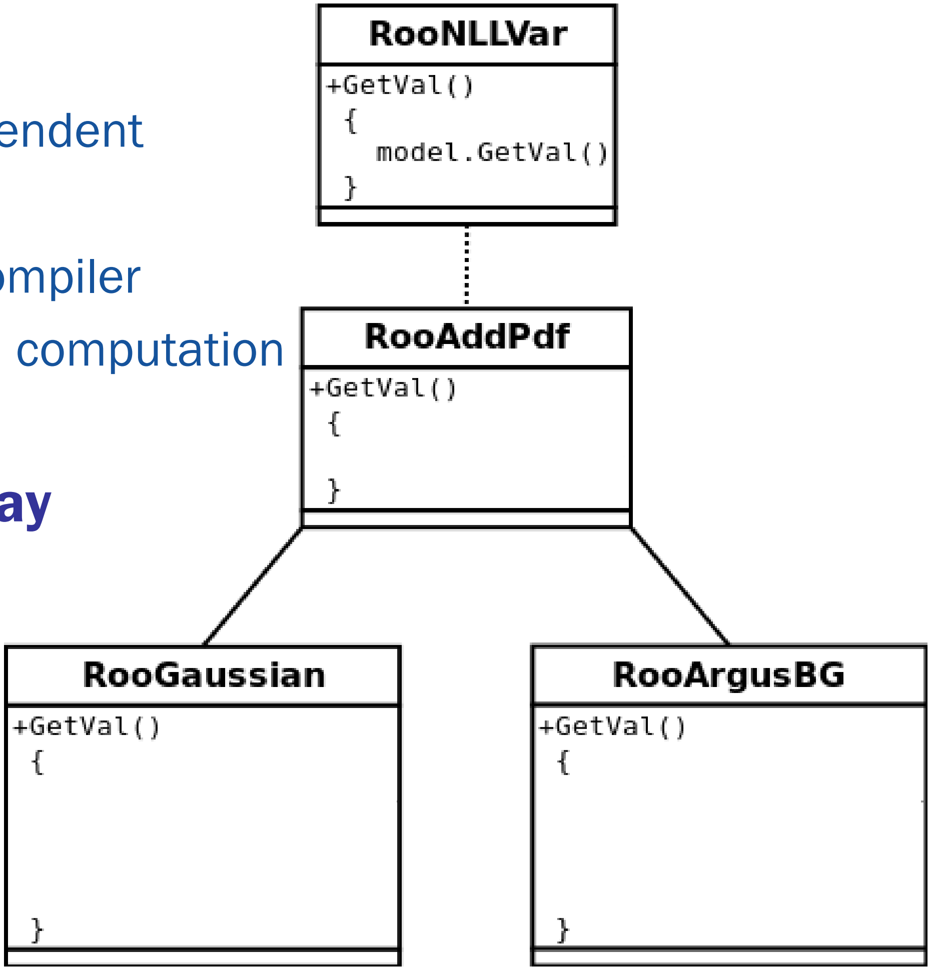
- G: **Gaussian**
- A: **Argus function**
- P: **Polynomial**

~40-50% of the execution time is spent in exp's calculation

Note: all PDFs have analytical normalization integral, i.e. >98% of the sequential portion can be parallelized

OpenMP parallelization

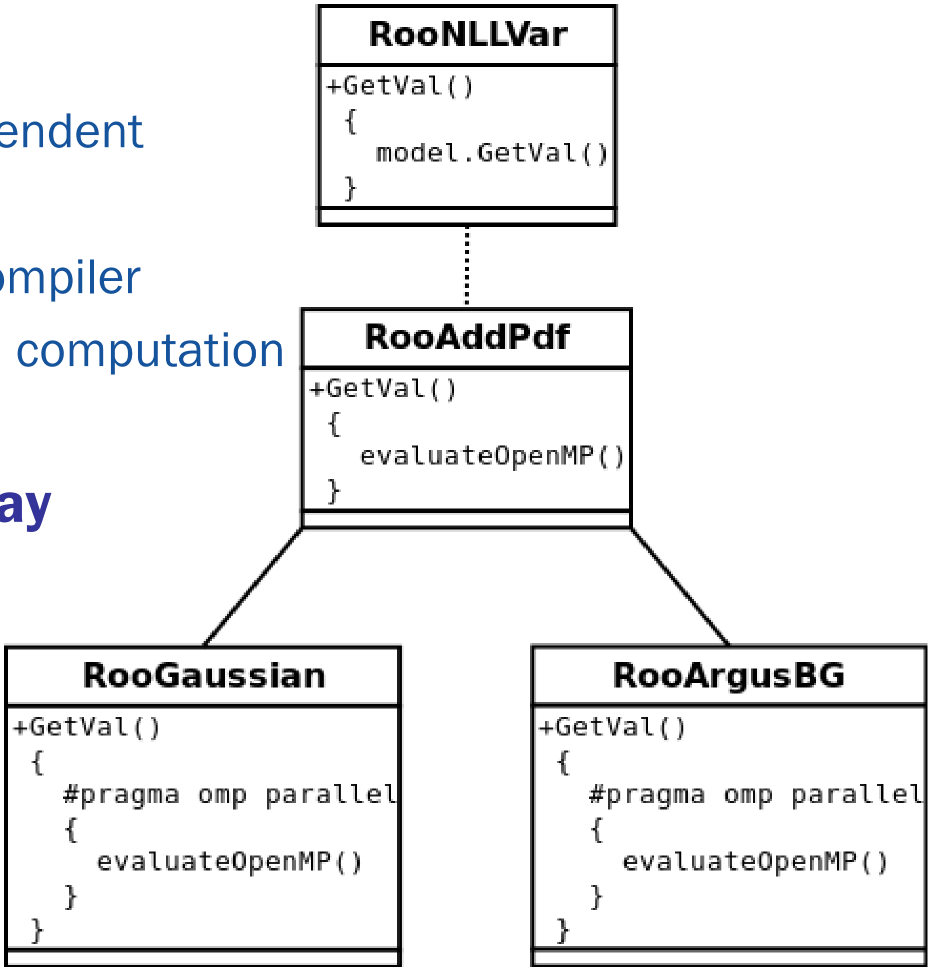
- **Instead of ”polling” the tree one value after another, do a whole range inside**
- **This makes**
  - the number of virtual function calls independent of N
  - the code ”SIMD-friendly”, i.e. easier for compiler to vectorize since we now have loops with computation
- **Downside: have to keep one entire array of results <u>per PDF</u> in memory until final NLL value is produced**



OpenMP parallelization

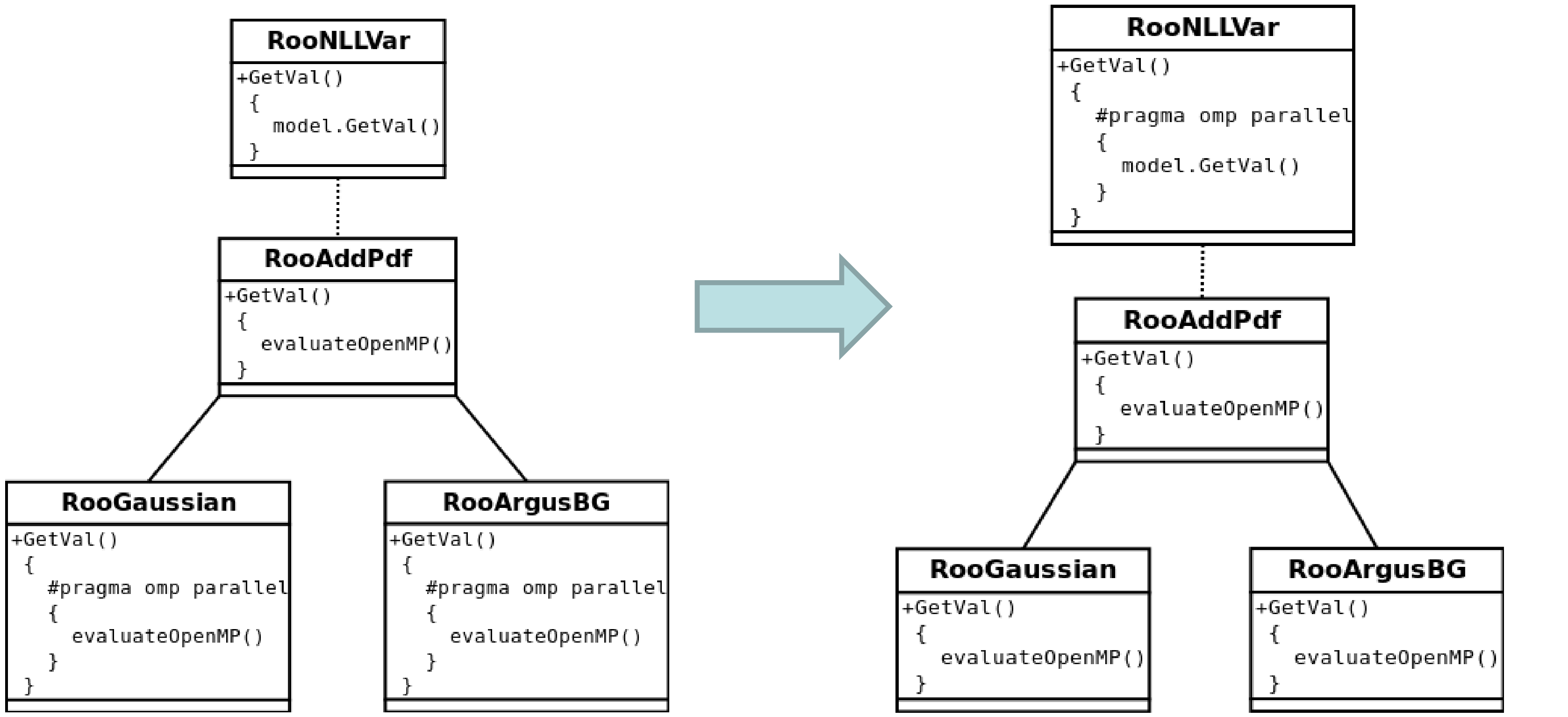
- **Instead of ”polling” the tree one value after another, do a whole range inside**
- **This makes**
  - the number of virtual function calls independent of N
  - the code ”SIMD-friendly”, i.e. easier for compiler to vectorize since we now have loops with computation

- **Downside: have to keep one entire array of results <u>per PDF</u> in memory until final NLL value is produced**



Optimizations

- First of all, get rid of all the parallel regions (minimize overhead)



```
RooNLLVar
+GetVal()
 {
   model.GetVal()
 }

RooAddPdf
+GetVal()
 {
   evaluateOpenMP()
 }

RooGaussian
+GetVal()
 {
   #pragma omp parallel
   {
     evaluateOpenMP()
   }
 }

RooArgusBG
+GetVal()
 {
   #pragma omp parallel
   {
     evaluateOpenMP()
   }
 }
```

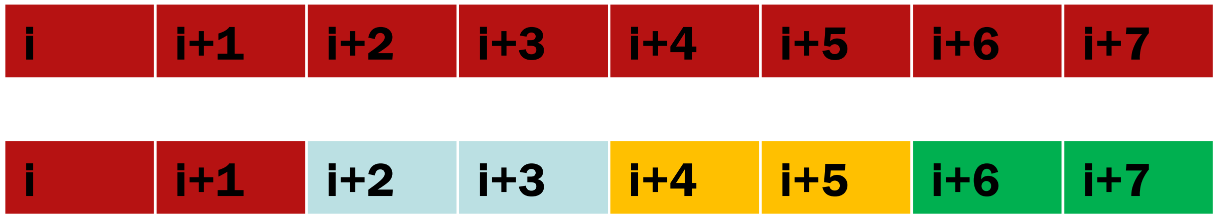
```
RooNLLVar
+GetVal()
 {
   #pragma omp parallel
   {
     model.GetVal()
   }
 }

RooAddPdf
+GetVal()
 {
   evaluateOpenMP()
 }

RooGaussian
+GetVal()
 {
   evaluateOpenMP()
 }

RooArgusBG
+GetVal()
 {
   evaluateOpenMP()
 }
```

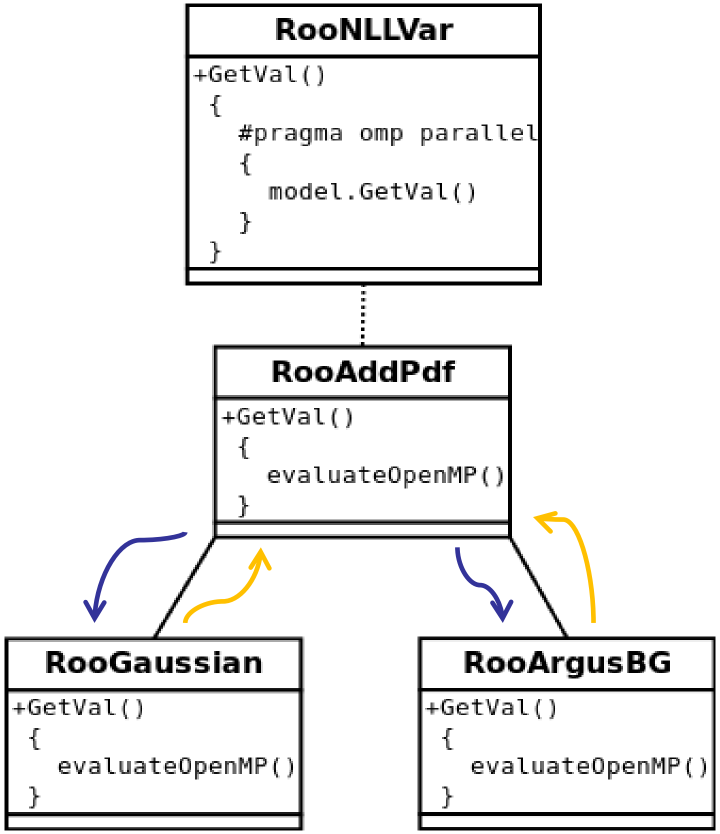
"Explicitly" parallel evaluation *"Implicitly" parallel evaluation*

- Improves performance and reduces OpenMP code to a few lines. Not without downsides though; harder to program/debug and makes it easier to introduce race conditions

Further optimizations

- Tests have shown a significant memory hotspot in composite PDFs (for a commodity Intel processor), preventing good scalability. Therefore we do cache blocking and ”result propagation”.



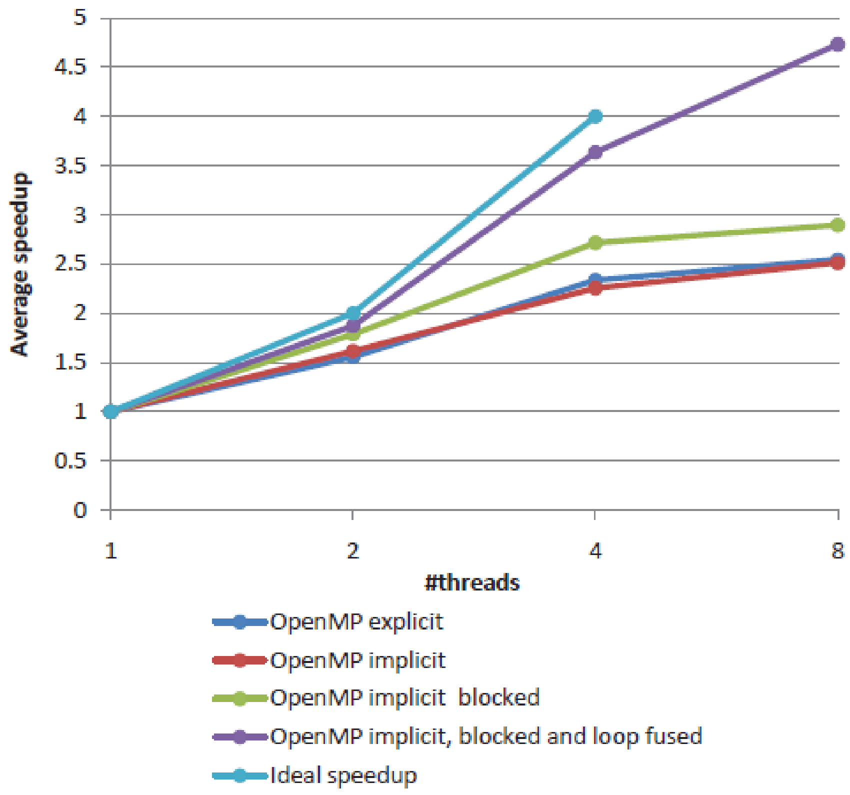


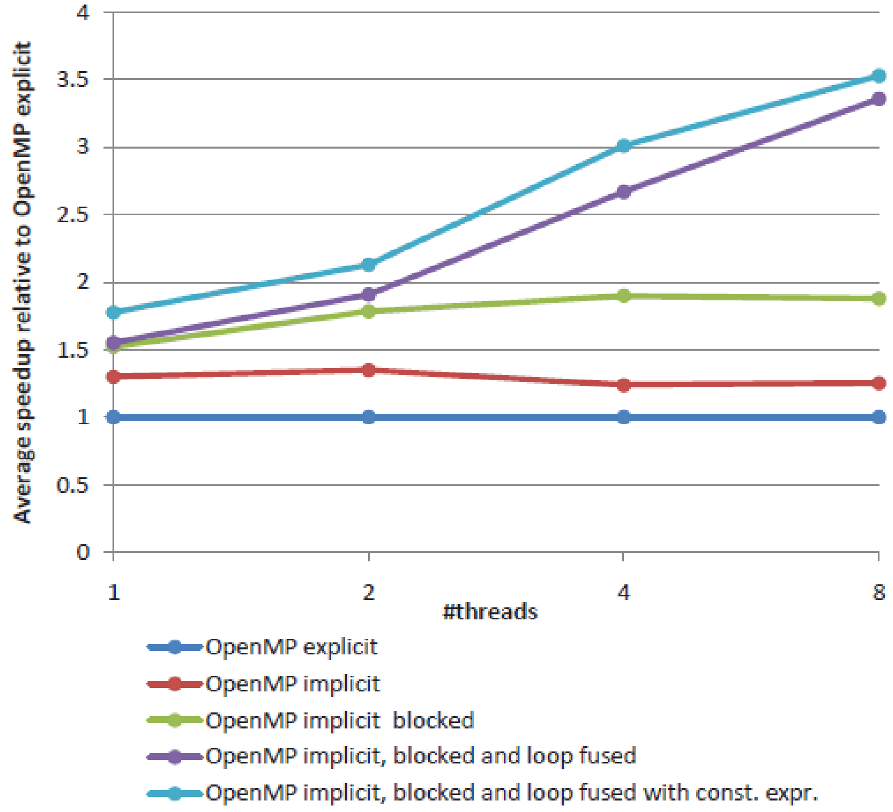
```
RooNLLVar
+GetVal()
 {
   #pragma omp parallel
   {
     model.GetVal()
   }
 }

RooAddPdf
+GetVal()
 {
   evaluateOpenMP()
 }

RooGaussian
+GetVal()
 {
   evaluateOpenMP()
 }

RooArgusBG
+GetVal()
 {
   evaluateOpenMP()
 }
```

- Eliminates memory hotspots and reduces memory requirements substantially (stores results only for composite nodes)
- In addition we precalculate expressions which are guaranteed to be constant during the evaluation (as opposed to before)

Scalability and performance

Average numbers with 10k, 50k, 100k, 500k and 1M events. Intel C++ compiler.





Intel Core i7 965 3.2 GHz (Nehalem). 8 MB L3 cache. 4 cores supporting SMT

- **"OpenMP explicit" is ~4.5x faster than the original RooFit on a single core**
- **The new version is ~1.75x faster than OpenMP explicit, which makes it in average ~7.8x faster. On top comes a scalability of ~3.6x with 4 threads and ~4.7x with 8 SMT threads. No increase in memory footprint w.r.t. #threads.**

MLFit and GPUs

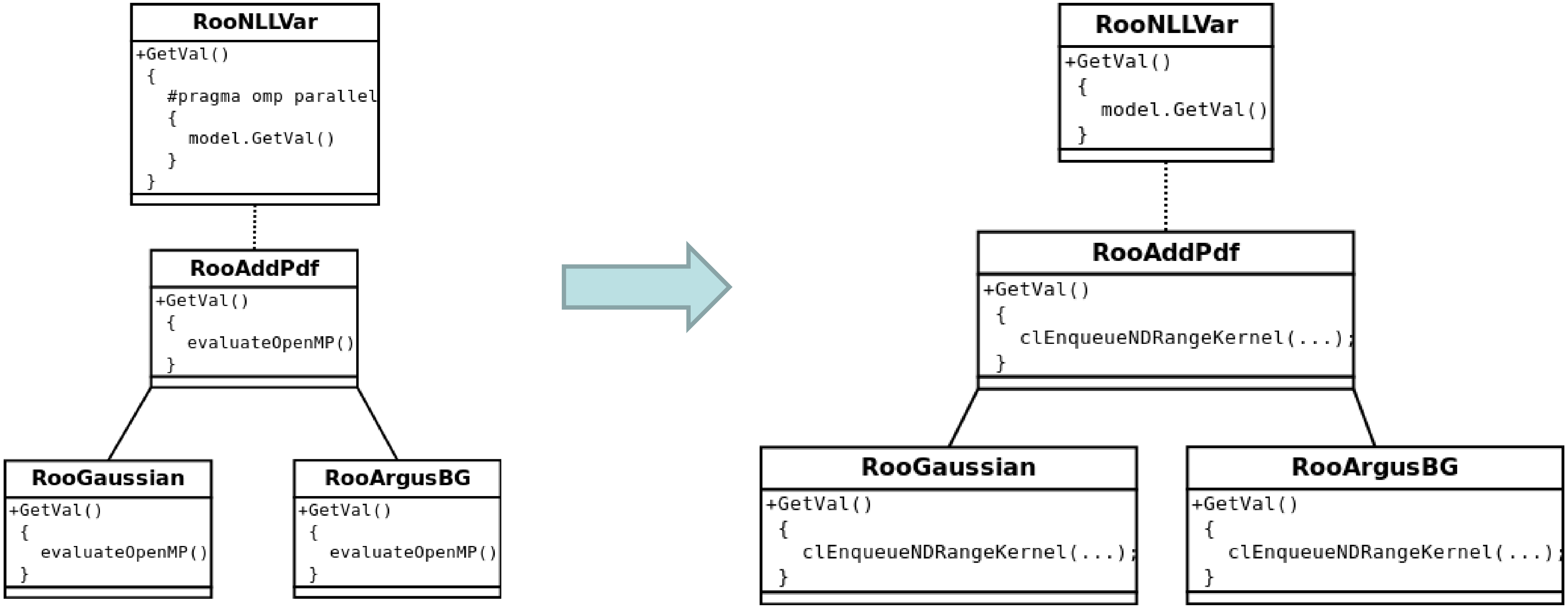
- OpenCL is a standard for heterogeneous computing set by the Khronos group (many significant industry leaders)
- Wanted to try OpenCL to target both NVIDIA and AMD GPUs
- The OpenCL idea: implicit data-parallel code executed in ”kernels”, portable across different devices/vendors

```
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);
  }
}
```

```
 __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);
}
```

MLFit and GPUs

- Implicitly parallel evaluation is tedious with OpenCL, since it is a 2nd environment (in addition to the program itself). This means:

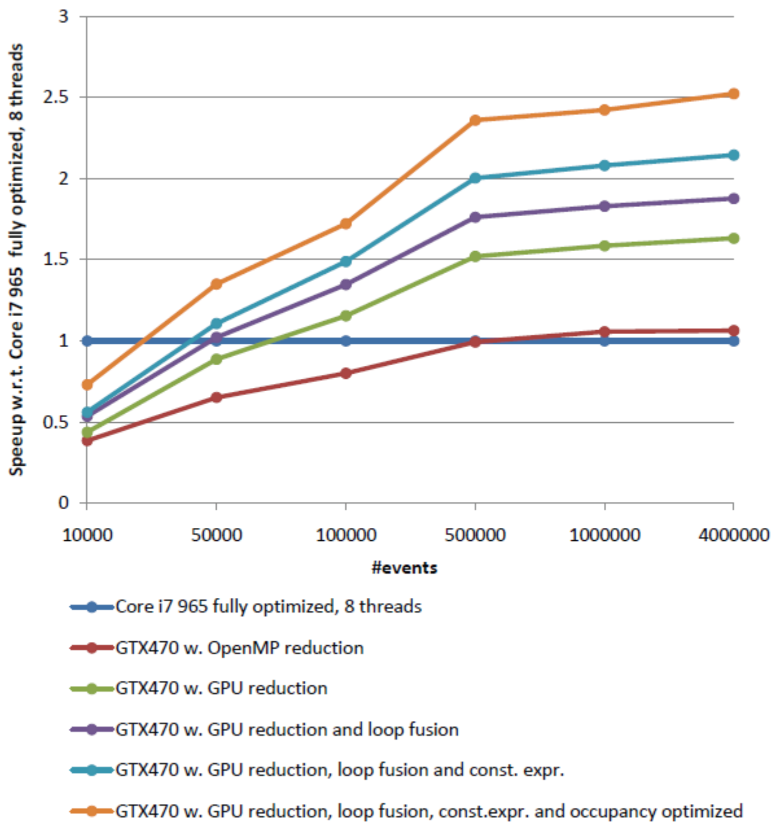


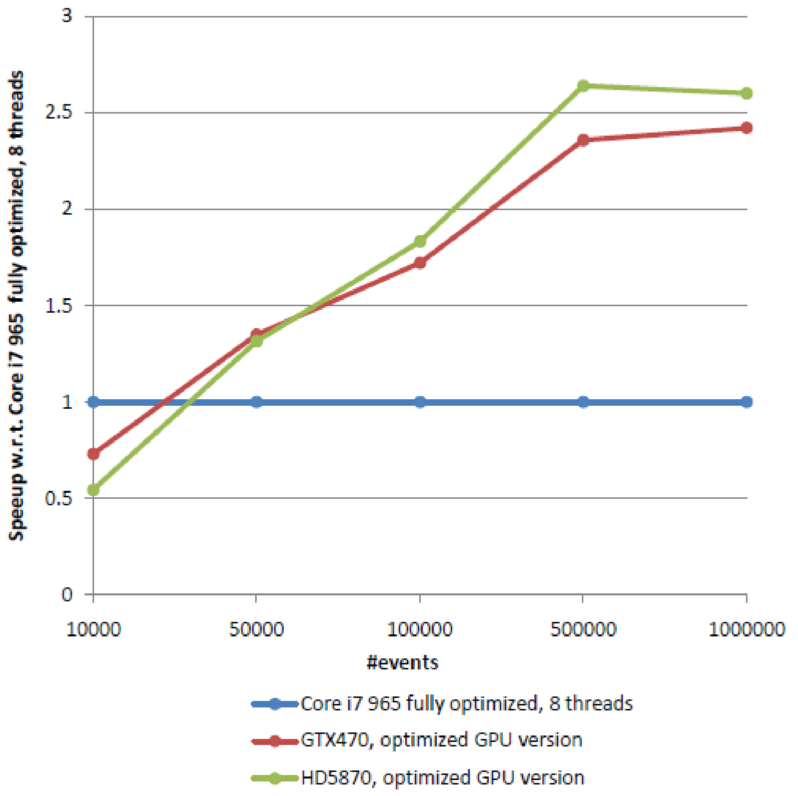
- Important to note that the CPU will now do a bit of work while walking the tree (might act as a bound for the GPU)

GPU optimizations

- Single-precision difficult because of minimizers, code base and result accuracy
- Added parallel reduction on the GPU. Means transferring a constant amount of values over the bus
- Double precision means no texture cache possibilities
- Fusing the normalization loop and using constant expressions also here
- Tuning workgroup sizes to get a decent occupancy gives significant improvements. We use a simple ”manual heuristic” for this
- All in all, a very limited case for GPU optimization
- In the results on the next slide we use two GPUs; NVIDIA GTX470 and AMD Radeon HD5870 (+ the i7 965 CPU from the previous results)

Results





- HD5870 has theoretically 4x as much computing power as the GTX470 when doing double-precision arithmetic (but costs ~ the same)
- We have done tests with simpler test kernels which show that arithmetic intensity must be enormous to exploit the HD's additional performance

Hybrid implementation

- Interesting to explore how to exploit all computational devices (CPUs and GPUs) fully, atleast in these Fusion days
- We have tested OpenCL on CPUs, and to make a long story short, it is in our case neither performant nor elegant compared to auto-vectorizing compilers and OpenMP
- We therefore want to use OpenMP + OpenCL in a hybrid scenario

Strategy and implementation

- Tedious to use task-based dynamic load balancing and still forcing determinism
- A priori static balancing will most probably be highly sub-optimal
- We want to do a self-refining static balancing in the start, reach convergence, and use that for the actual fit.
- We start with equal partitions. An updated set of partitions is based on the execution time of each device
- i.e.

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

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

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

- Method partly based on Galindo et al.: 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.

Strategy and implementation

- ❑ **But what about threading and all that?**

```
int OMP_NUM_THREADS = omp_get_max_threads();
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
}
```

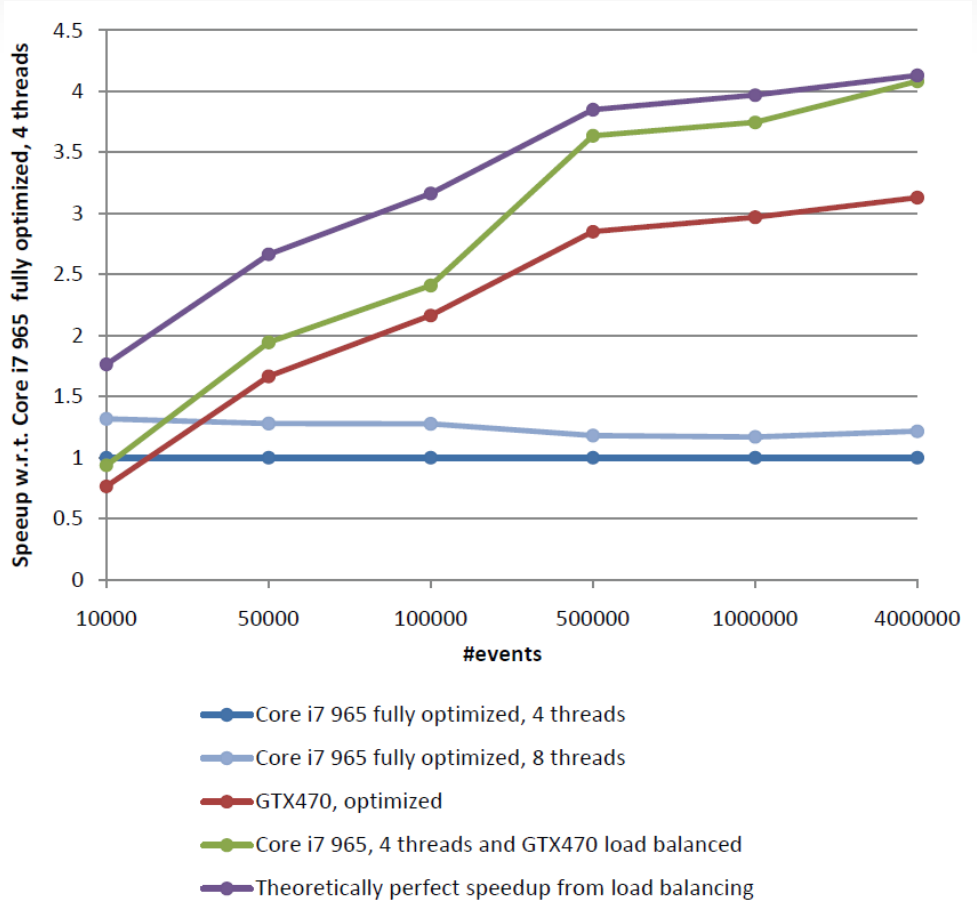
- ❑ **We spawn one thread per GPU in addition to any threads that run CPU computation**
- ❑ **The tree-walking for the CPU thread responsible for GPU execution should therefore ideally impose minimal overhead**
- ❑ **This effect of course diminishes as the number of cores grow (probably more ideal to use on a 10-core processor than on a 4-core)**

Results

- First of all, SMT does not contribute anything anymore

- The potential is clearly illustrated in this plot. Balancing is as good as perfect when N grows high enough

- Timings of the GTX470 has been extr-emely accurate with low deviation



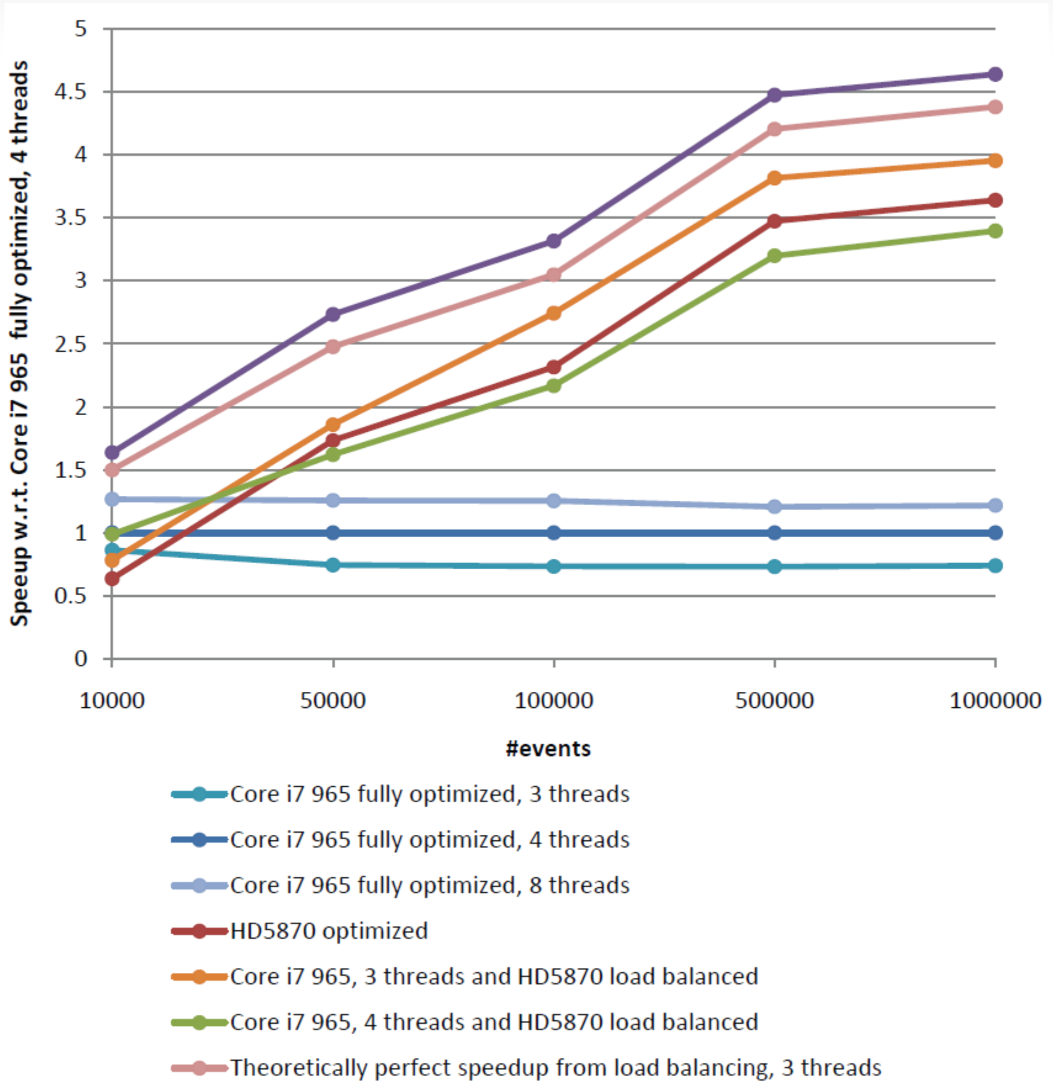
CERN openlab

Results cont.

- ❑ AMDs OpenCL implementation incurs larger overhead and HD5870 timings have higher deviation

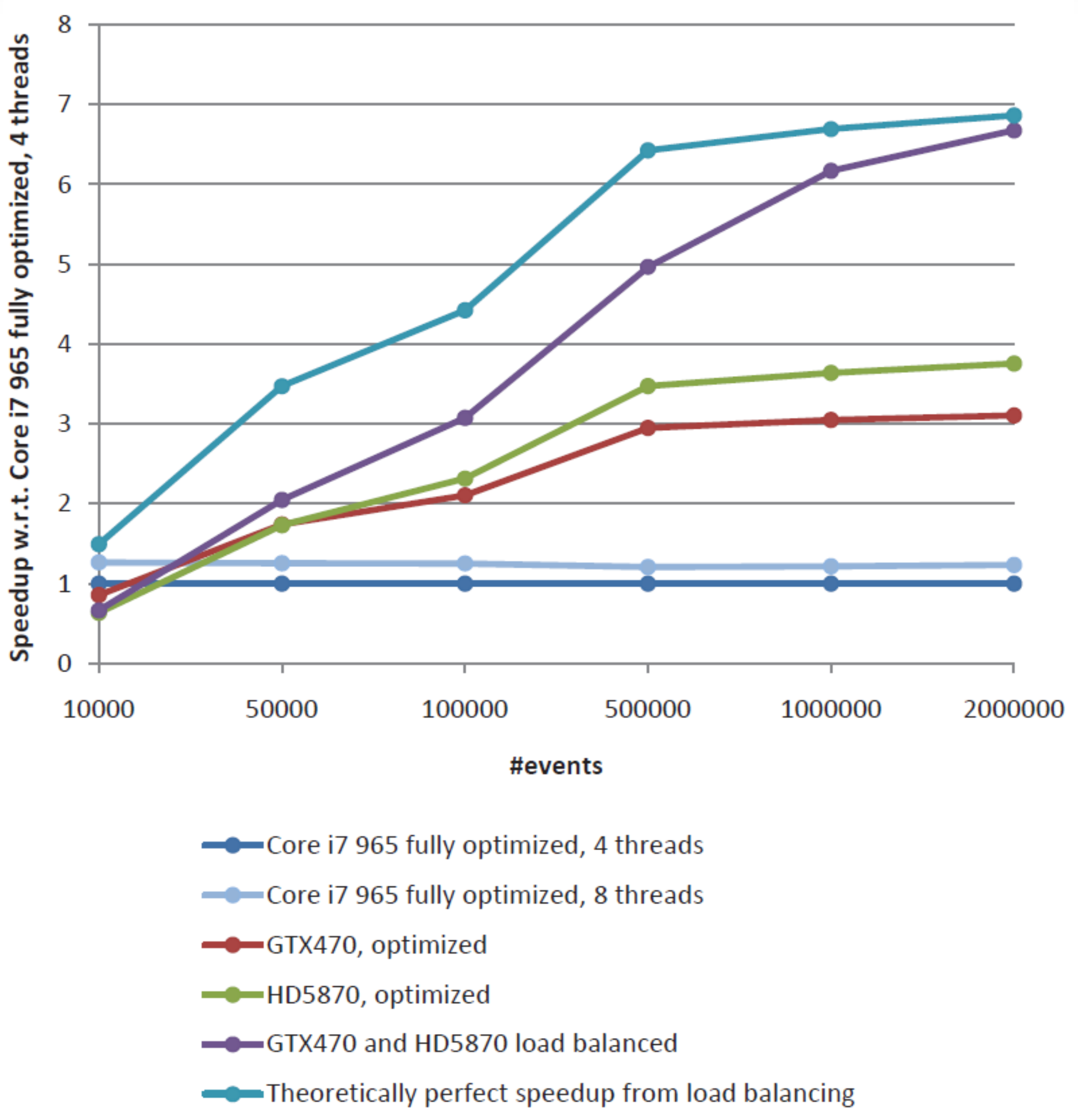
- ❑ We have to use 3 computational CPU threads instead of 4 to actually gain something

- ❑ ..but the gain is almost negligible. In other words, this is a non-ideal case



Results cont.

- Multi-GPU solution works ideal when N grows large
- Note that GPU potential is lowe-red when doing less work (and that happens when we now divide)



Conclusions

- In every case, find out if you are compute-bound or memory-bound first!
- OpenMP and OpenCL can co-exist fairly well. However, CUDA can be a lot more suitable for large C++ programs (e.g. code reuse), and can also inflict on performance by using C++ features (templates is a good example)
- Low/negligible OpenCL API overhead and device timing accuracy is paramount for the hybrid implementation to work good
- When that is satisfied, it is an effective data-parallel approach to exploit e.g. Fusion APUs from AMD, when results must be guaranteed reproduceable (very difficult, if not practically impossible with task-based dynamic load balancing)
- Might seem obvious, but devices should perform comparably. No point in balancing e.g. a 7:1 ratio case