Maximum Likelihood Fits on GPUs

S. Jarp, A. Lazzaro, J. Leduc, A. Nowak, F. Pantaleo
CERN openlab

International Conference on Computing in High Energy and Nuclear Physics 2010 (CHEP2010)

October 21st, 2010

Academia Sinica, Taipei

Presented by Alfio Lazzaro

Maximum Likelihood Fits

- We have a sample composed by *N* events, belonging to *s* different species (signals, backgrounds), and we want to extract the number of events for each species and other parameters
- We use the *Maximum Likelihood fit technique* to estimate the values of the free parameters, minimizing the Negative Log-Likelihood (*NLL*) function

$$NLL = \sum_{j=1}^{s} n_j - \sum_{i=1}^{N} \left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j(x_i; \theta_j) \right)$$

j species (signals, backgrounds)
n_j number of events
$\mathcal{P}_j$ probability density function (PDF)
θ_j Free parameters in the PDFs

MINUIT

- Numerical minimization of the *NLL* using MINUIT (F. James, *Minuit, Function Minimization and Error Analysis*, CERN long write-up D506, 1970)
- MINUIT uses the gradient of the function to find local minimum (MIGRAD), requiring
  - The calculation of the gradient of the function for each free parameter, naively

$$\left.\frac{\partial NLL}{\partial \hat{\theta}}\right|_{\hat{\theta}_0} \approx \frac{NLL(\hat{\theta}_0 + \hat{\mathrm{d}}) - NLL(\hat{\theta}_0 - \hat{\mathrm{d}})}{2\hat{\mathrm{d}}}$$

2 function calls per each parameter

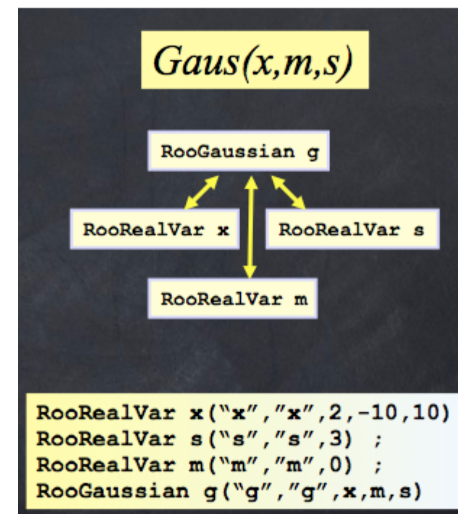
  - The calculation of the covariance matrix of the free parameters (which means the second order derivatives)
- The minimization is done in several steps moving in the Newton direction: each step requires the calculation of the gradient

⇨ Several calls to the *NLL*

Building models: RooFit

- RooFit is a Maximum Likelihood fitting package (W. Verkerke and D. Kirkby) for the *NLL* calculation
  - Inside ROOT (details at http://root.cern.ch/drupal/content/roofit)
  - Allows to build complex models and declare the likelihood function
  - Mathematical concepts are represented as C++ objects

Mathematical concept		RooFit class
variable	x	RooRealVar
function	$f(x)$	RooAbsReal
PDF	$f(x)$	RooAbsPdf
space point	$\vec{x}$	RooArgSet
integral	$\int_{x_{min}}^{x_{max}} f(x)dx$	RooRealIntegral
list of space points		RooAbsData

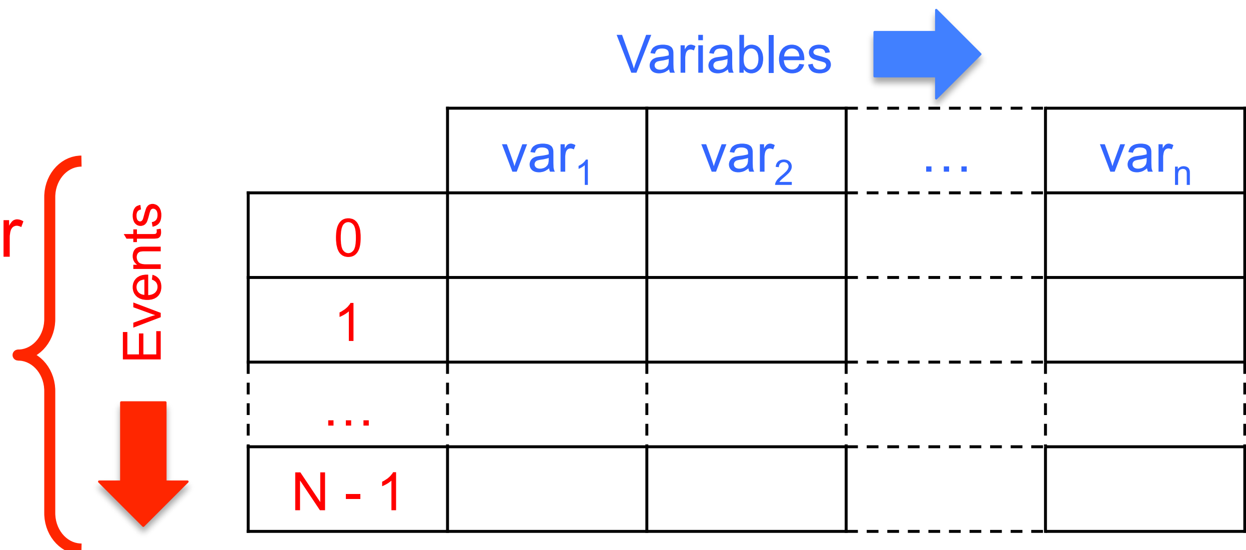


- On top of RooFit developed another package for advanced data analysis techniques, RooStats
  - Limits and intervals on Higgs mass and New Physics effects

Likelihood Function calculation in RooFit

1. Read the values of the variables for each event
2. Make the calculation of PDFs for each event
   - Each PDF has a common interface declared inside the class RooAbsPdf with a virtual method `evaluate()` which define the function
     - Each PDF implements the method `evaluate()`
   - Automatic calculation of the normalization integrals for each PDF
   - Calculation of composite PDFs: sums, products, extendend PDFs
3. Loop on all events and make the calculation of the *NLL*

Parallel execution over the events (as it is already implemented)



Algorithms

- Two algorithms implemented:
  1. RooFit Event-based (CPU Implementation), described before
     - Parallelization at event level, using fork
     - Not shared resources
  2. PDF-Event-based Algorithm
     - GPU Implementation (CUDA)
     - CPU Implementation (OpenMP)

Note: everything done in double precision

CERN openlab

PDF-Event-based Algorithm

New approach to the *NLL* calculation:

1. Read all events and store in arrays in memory
2. For each PDF make the calculation on all events
   - Corresponding array of results is produced for each PDF
   - Evaluation of the function inside the local PDF, i.e. not need a virtual function (drawback: require more memory to store temporary results: 1 double per each event and PDF)
   - Apply normalization
3. Combine the arrays of results (composite PDFs)
4. Calculation of the *NLL*

Parallelization splitting calculation of each PDF over the events

- Particularly suitable for thread parallelism on GPU, requiring one thread for each PDF/event
- Possible benefit from vectorization on the CPU

Test environment

- PCs
  - CPU: Nehalem @ 3.2GHz: 4 cores – 8 hw-threads
  - OS: SLC5 64bit - GCC 4.3.4
  - ROOT trunk (October 11th, 2010)
- GPU: ASUS nVidia GTX470 PCI-e 2.0
  - Commodity card (for gamers)
  - Architecture: GF100 (Fermi)
  - Memory: 1280MB DDR5
  - Core/Memory Clock: 607MHz/837MHz
  - Maximum # of Threads per Block: 1024
  - Number of SMs: 14
  - CUDA Toolkit 3.1 06/2010
  - Developer Driver 256.40
  - Power Consumption 200W
  - Price ~$340

CERN openlab

PDFs implemented

- 1D PDFs commonly used in HEP:
  - Symmetric and Asymmetric Gaussian
  - Breit-Wigner
  - Crystal Ball Function
  - Argus
  - Generic Polynomial
  - Chi Square
- Composition of PDFs:
  - Sum of two or more PDFs
  - Product of two or more PDFs
  - Multivariate PDFs
- Very easy to build complex models (via composition) and add new PDFs

GPU Implementation

- Data are copied on the GPU once
- Results for each PDF are resident only on the GPU
  - Arrays of results are allocated on the global memory once and they are deallocated at the end of the fitting procedure
    - Minimize CPU ⇔ GPU communication
  - Only the final results are copied on the CPU for the final sum to compute *NLL*

- Device algorithm performance with a linear polynomial PDF and 1,000,000 events
  - 45 GFLOPS and 3.5 GB/s CPU ⇔ GPU data transfer

1D PDF Tests

1,000,000 events and 1000 iterations

PDF Name	Formula	CPU vs GPU time ratio	kernels execution time portion
Gaussian	$\exp\{-\frac{(x-\mu)^2}{2\sigma^2}\}$	11.3	21.3%
Asymmetric Gaussian	$\begin{cases} \exp\{-\frac{(x-\mu)^2}{2\sigma_l^2}\} & x \leq \mu \\ \exp\{-\frac{(x-\mu)^2}{2\sigma_r^2}\} & x > \mu \end{cases}$	11.9	21.9%
Breit-Wigner	$\frac{1}{(x-x_0)^2+\Gamma^2/4}$	2.4	13.4%
χ^2	$\frac{1}{2^{k/2}\Gamma(\frac{k}{2})} x^{\frac{k}{2}-1} \exp\{-x/2\}$, $k=5$	27.6	70.0%
Argus	$\sqrt{\left(1-\frac{x^2}{c^2}\right)} \exp\left\{-\frac{1}{2}\eta\left(1-\frac{x^2}{c^2}\right)\right\}$	25.3	50.1%

- CPU algorithm is the event-based (RooFit) in sequential
- GPU time includes data transfer time (data and results)
  - A significant portion of time, limiting the scalability
  - More complex PDF => Bigger portion of time spent in evaluation VS time for data transfers

CERN openlab

Complex Model Test

$$
\begin{aligned}
n_a[f_{1,a}G_{1,a}(x) + (1 - f_{1,a})G_{2,a}(x)]AG_{1,a}(y)AG_{2,a}(z) +& \\
n_b G_{1,b}(x) BW_{1,b}(y) G_{2,b}(z) +& \\
n_c AR_{1,c}(x) P_{1,c}(y) P_{2,c}(z) +& \\
n_d P_{1,d}(x) G_{1,d}(y) AG_{1,d}(z)&
\end{aligned}
$$

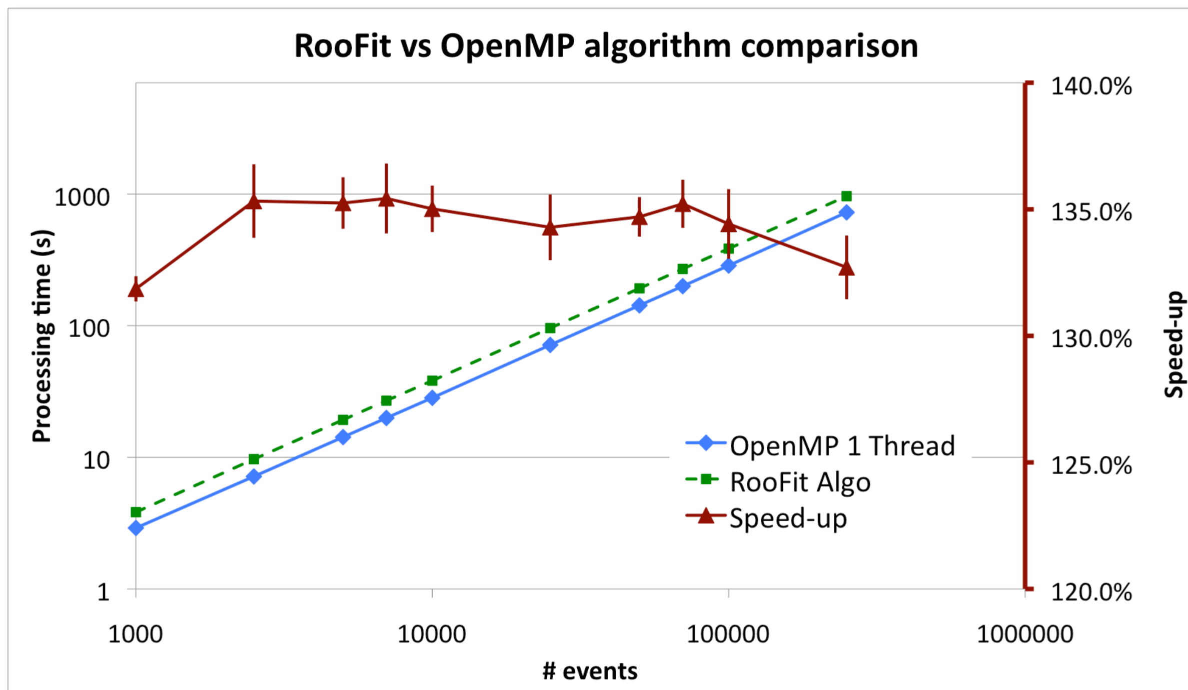
17 PDFs in total, 3 variables, 4 components, 35 parameters

- G: Gaussian
- AG: Asymmetric Gaussian
- BW: Breit-Wigner
- P: Polynomial

Note: all PDFs have analytical normalization integral

Event-based VS PDF-event-base performance

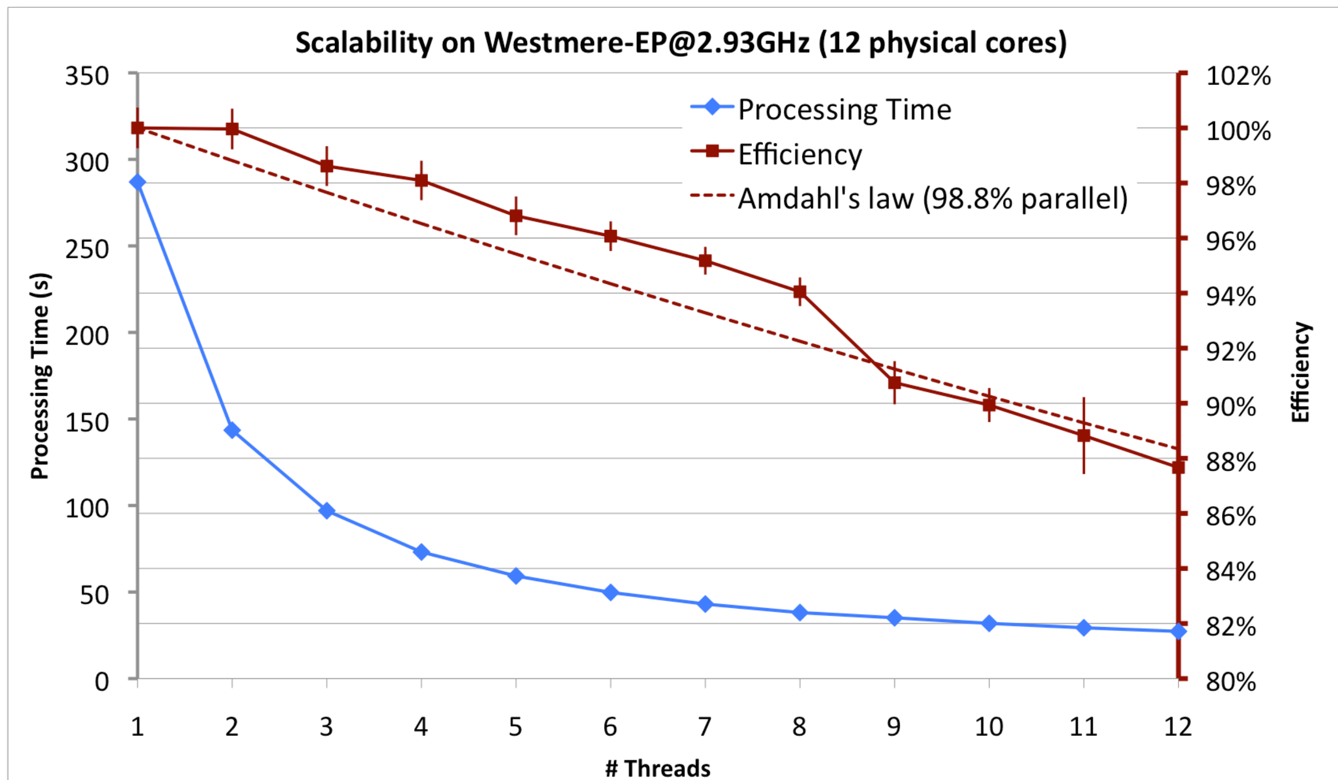
- Driven by the GPU implementation, we implemented a corresponding CPU implementation
  - take benefit from the code optimizations (due to migration from C++ to C)
    - No virtual functions
    - Inlining of the `evaluate` function
    - Data organized in C arrays, perfect for vectorization
  - it can be easily parallelized using OpenMP



- Linear increase with the number of events (as expected)
- Speed-up of 34% (almost flat over the number of events), just optimizing the algorithm! (not parallelization)

PDF-event-base scalability with OpenMP

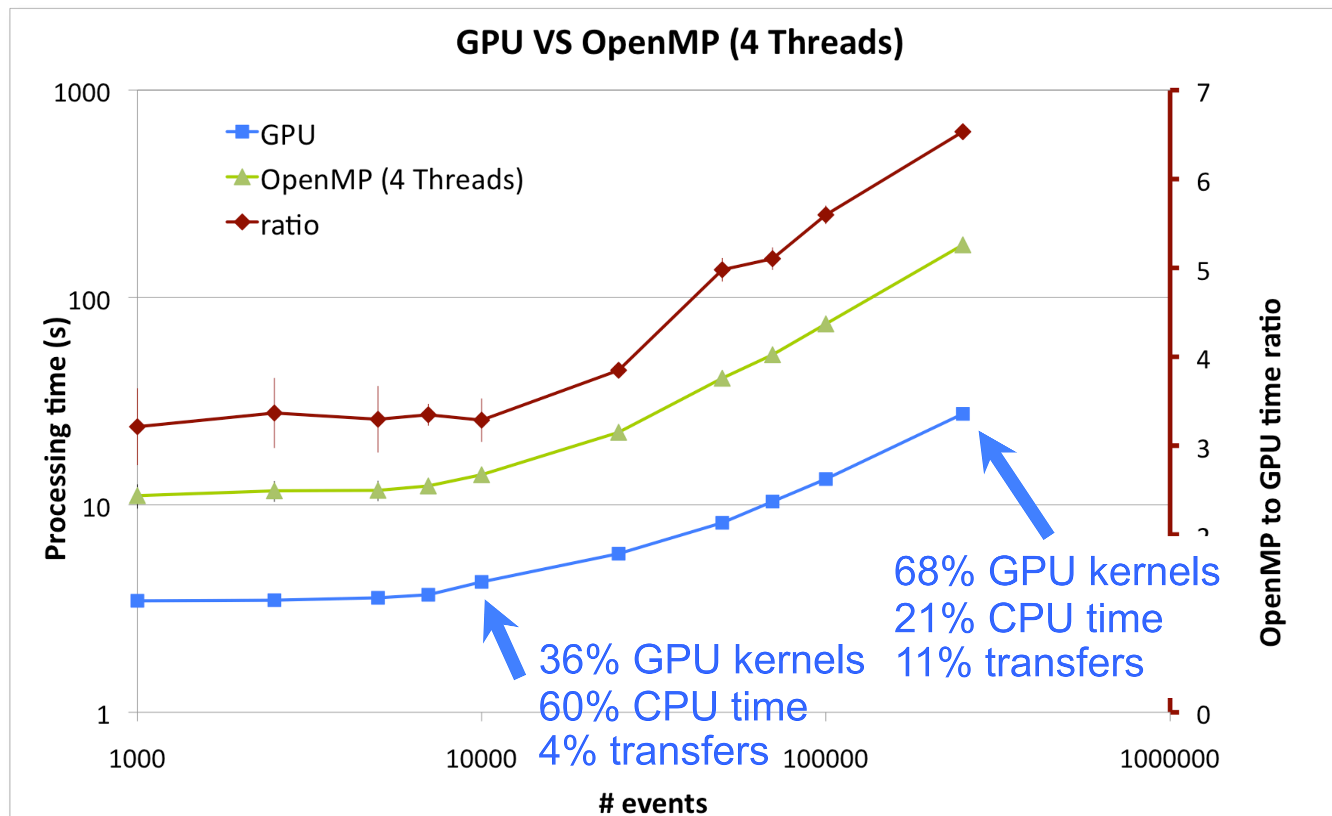
- Test done on the Westmere-EP @ 2.93 GHz
  - 12 cores / 24 threads
- 100,000 events
- 98.8% of the sequential execution can be parallelized (1.2% required for initialization of the arrays for data and results and normalization integrals calculation)



- Negligible increase in memory (arrays are shared)
- Scalability as expected
- Using SMT (hw-threading) with 24 threads we reach 110% in efficiency w.r.t 12 threads (+32% in case of ideal speed-up)

PDF-event-base: GPU VS OpenMP

- Fair comparison
  - Same algorithm
  - Algorithm on CPU optimized and parallelized (4 threads)
  - CPU does the final sum of the *NLL* and normalization integral calculations
- Check that the results are compatible: asymmetry less than 10^{-12}



- Speed-up increases with the dimension of the sample, taking benefit from the data streaming on GPU and the integral calculation only on the CPU
  - ~3x for small samples, up to ~7x for large samples

Conclusion

- ✓ Implementation of the algorithm in CUDA to calculate the *NLL* on GPU, as part of the RooFit package
  - ❑ Require not so drastic changes in the existing RooFit code
  - ❑ New design of the algorithm for PDF-event parallelism
- ✓ The CUDA implementation “forces” us to develop an OpenMP implementation on the CPU of the same PDF-event algorithm
  - ❑ With 1 thread +34% better performance with respect to RooFit implementation
- ✓ In our test GPU implementation gives >3x speed-up (~7x for large samples) with respect to OpenMP with 4 threads
  - ❑ Note that our target is running fits at the user-level on the GPU of small systems (laptops), i.e. with small number of CPU cores
- ✓ This is a preliminary work (mainly by the summer student, Felice). Still a lot to do. Some examples:
  - ❑ Simultaneous fits with index variables
  - ❑ More complex tests
  - ❑ Parallelization of PDFs with numerical integrals
  - ❑ Further optimization on the GPU (better treatment of the memory)
- ✓ Last but not least: insert the code in the official RooFit/ROOT release

Backup Slides

CUDA inside RooFit

CPU

```
Double_t Pdf::evaluate() const // virtual method
{
    return <function to be evaluated>(<data>,<pars>); // data and pars are data members of the Pdf Class
}
```

GPU

```
Double_t Pdf::evaluate() const // virtual method
{
    return evaluateLocal(<data>,<pars>);  // data and pars are data members of the Pdf Class
}

//inside class definition
#ifdef __USECUDA__
  friend void KernelEvaluatePdf(const Pdf* pdf, const Double_t* data,
        <par>, Double_t *results, const UInt_t N);

  __host__ __device__
#endif
inline Double_t Pdf::evaluateLocal(<data>,<pars>) const // inline method
{
    return <function to be evaluated>(<data>,<pars>); // data and pars are local to the method
}
```

CUDA inside RooFit

GPU code

```
#ifdef __USECUDA__

Bool_t Pdf::evaluate(const Data& data) const // data links events on GPU and CPU
{
    // set pointer to GPU for the results of the evaluation
    ...
    // pars are local to the method
    KernelEvaluatePdf<<<NUM_BLOCKS,NUM_THREADS)>>>(this,<pointer to GPU data>,pars,<pointer to GPU results>,data.size());

    return kTRUE;
}

__global__ void KernelEvaluatePdf(const Pdf* pdf, const Double_t* data,
        <pars>, Double_t *results, const UInt_t N)
{
    UInt_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx<N) {
        results[idx] = pdf->evaluateLocal(data[idx],<pars>);
    }

}

#endif
```

RooMinimizer

- Interface to MINUIT: calculate the gradient of the NLL

$$\left.\frac{\partial NLL}{\partial \hat{\theta}}\right|_{\hat{\theta}} \approx \frac{NLL(\hat{\theta}+\hat{\mathrm{d}}) - NLL(\hat{\theta}-\hat{\mathrm{d}})}{2\hat{\mathrm{d}}}$$

2×(#pars) iterations

RooNLLVar

- Do the loop over the N events: $i=1 \dots N$
- For each event calculate the $\mathcal{P}$'s

$$NLL = \sum_{j=1}^{s} n_j - \sum_{i=1}^{N}\left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j(x_i;\theta_j)\right)$$

N iterations

RooAbsData

- Read the variables of an event i

RooAbsPdf

- Calculate the log term (**getLogVal** method)
- Evaluation of the function (public **getVal** virtual method)
- Propagate the evaluation of the function to all $\mathcal{P}$'s (through the specialized public **getVal** method of each $\mathcal{P}$)
  - Calculation of the function with the protected **evaluate** virtual method, defined for each $\mathcal{P}$
  - Normalization

$i=N$ — No / Yes

Gradient done — No / Yes