HPC MEM implementations at LLR

F. Beaudette, O. Davignon, G. Grasseau,
L. Mastrolorenzo, P. Paganini, T. Strebler

Laboratoire Leprince-Ringuet
CNRS/IN2P3, École Polytechnique

Motivation

- LLR involved in ME Method:
  Hττ, VBF (Luca M.) then (Thomas S.)
  ttH(→ττ) (Thomas S.)
- MEM CPU time consuming

  Estimating computation resources for integrations (20k points, 5 dimensions):
  - Run 1 (2010-2012, 7-8 TeV): ~200k ev → **4100 days** (full computation: 30 mn/ev) on 1 core
  - Run 2 (2015-2016, 13-14 TeV): **x 5**

- Challenge: great advantage for teams having an HPC implementation
- Re-usability: MC Integration widely used in HEP
- Hardware trends: use more and more large hardware vector arithmetics (Xeon Phi, GPGPU, AVX, SSE, ...)

LLR involved in HPC MEM based on new hardware

Outline

- MEM-MPI implementation
- Accelerating MEM analysis with new “many-cores” architectures (OpenCL)
- Inside OpenCL-MEM architecture
- Performance tests
- Conclusion & perspectives

HPC implementation

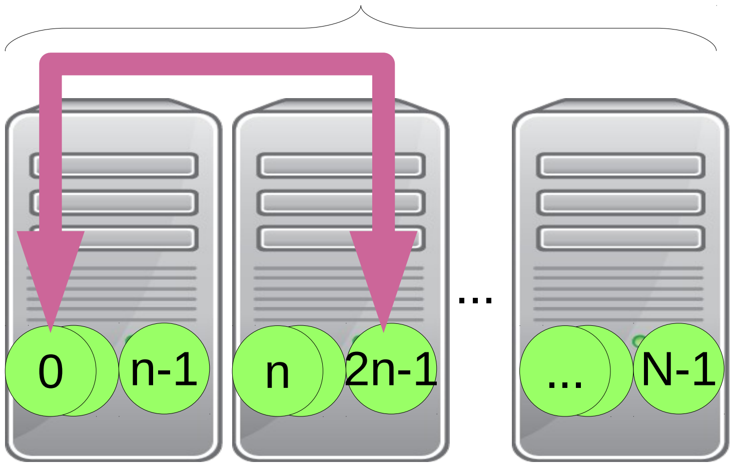
MPI implementation for production

- Motivation & context
- Technical choices
- MPI-MEM architecture: MPI Dispatcher
- Performance: scalability
- Possible improvements

MPI Implementation Context

- Luca and Thomas PhD
- Validate the numerical scheme (get rid of technical problems)
- Wrong way to design at first for **one** (two) accelerator(s) cards, need to **aggregate the computing power of several nodes**

N MPI processes running on several nodes



Message Passing Interface (MPI)

→ // Model: distributed memory

→ Point to point communications

→ Standard used on Supercomputers

→ **Deliver quickly an HPC implementation of MEM thanks to MPI**

MPI Implementation Technical choices

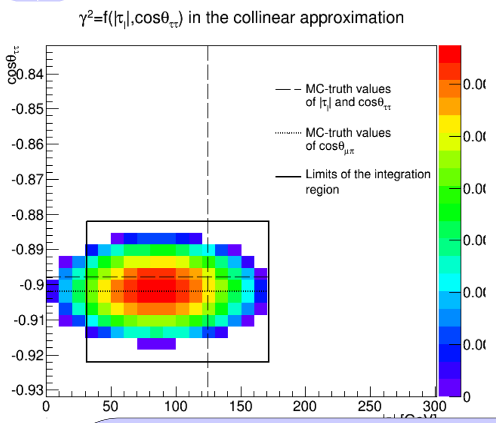
Overview

- Context: "easily" extensible to "many-core" platforms
- MPI: deployment on a large number of nodes
- MadGraph: more flexible to tackle ≠ channels (code generation)
- ROOT: I/O files, Lorentz algebra, histograms, LUT functions, ...
- LHAPDF: Parton Distribution Function 5.8.5 (6.x.x)

MC Integration

- VEGAS widely used in CMS collaboration (compared with others: Markov Chain MC, ...)
- GSL implementation :
  - Reference & not a black box,
  - The algorithm need to be well-known to // on GPU
  - Adjust the different parameters (# box, ...), improve the adaptability of the grid
  - Easy to change the RNG.

Choices made to facilitate "many-core" developments



MPI implementation

MPI Dispatcher

MPI ID = 0

Master

- Read the event from ROOT files
- Selects the event (Jet energy, μ/e, ...)
- Integral Boundaries computation (filter)
- Keep the order and (integration features) of the considered event (OrderedMap)
- Sends the integration features on a slave
- Receives a result message
- Update the OrderedMap
- Write the event results (with the same reading order)

MPI ID = N-1

MPI ID = 3

MPI ID = 2

MPI ID = 1

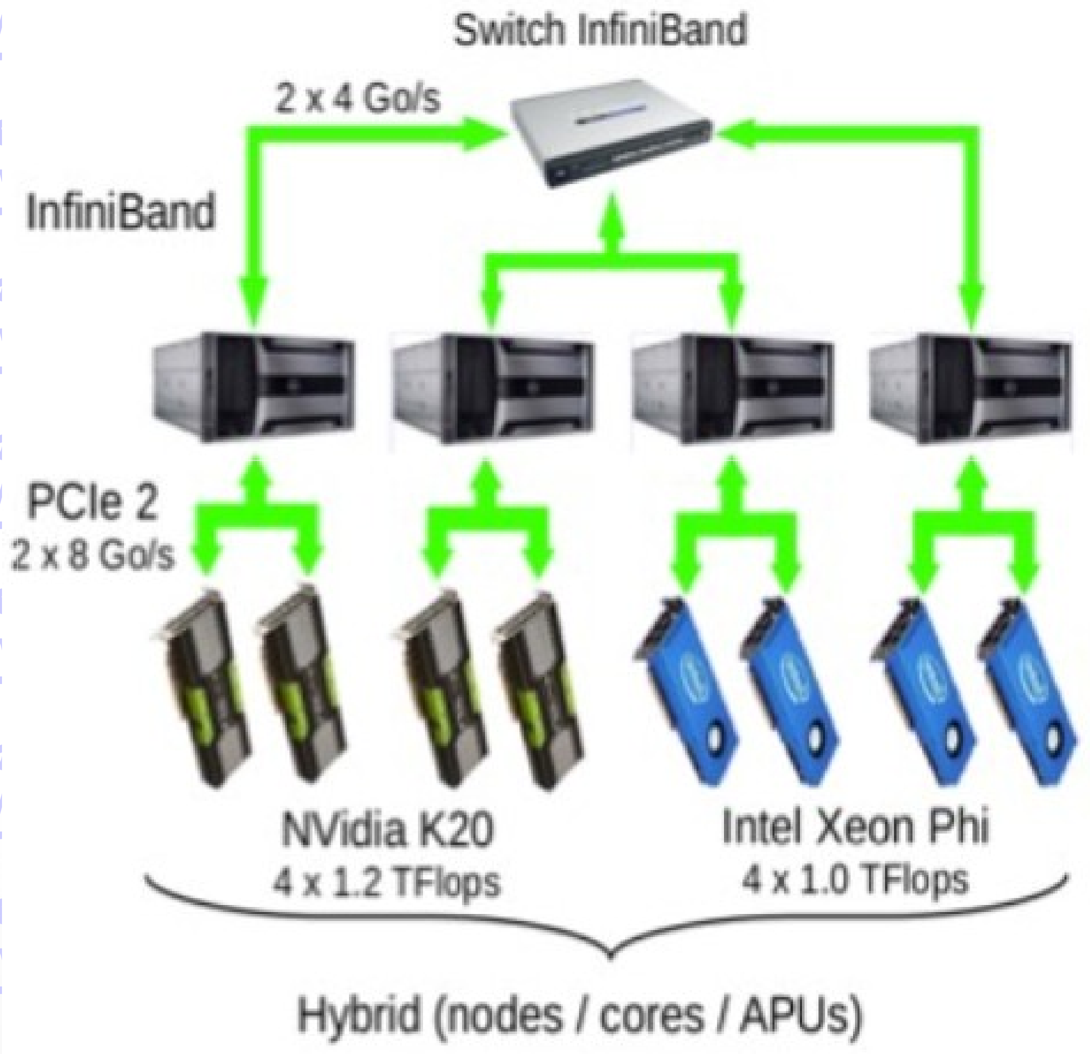
Slave/Worker

- Listen to new event(s) to compute
- Compute all integrals weight h_0, …, h_k (VBF, DY)
- Returns the integration results
- sends a ready message (load balancing)

A worker requests new computations as soon as it is idle
→ Expected **good load-balancing** (heterogeneous)

MPI Implementation

Benchmarks Platform: GridCL

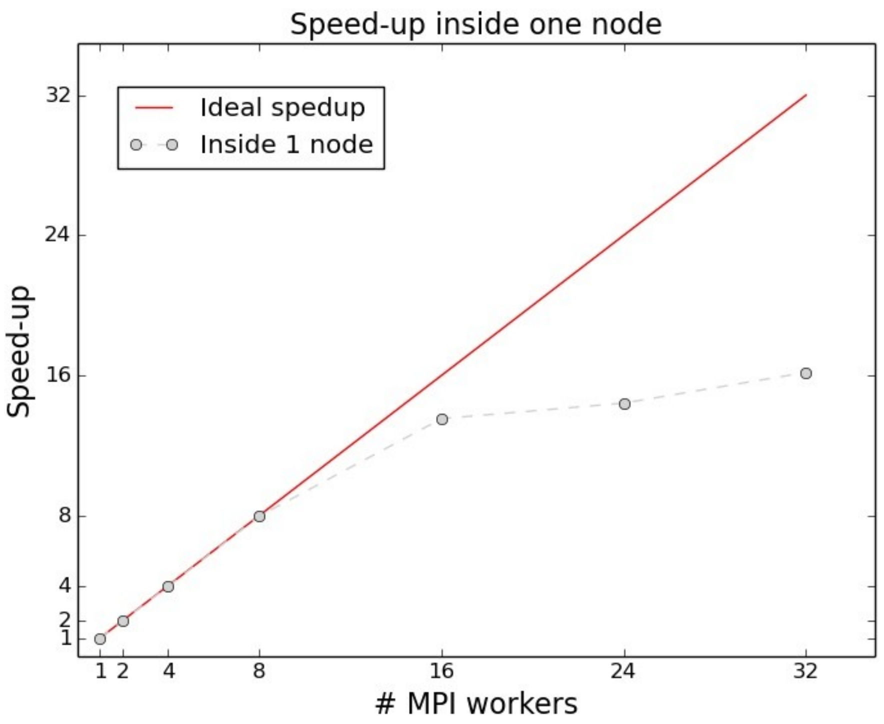


Each node

- 2 x Intel E5-2650: 2 GHz,16 cores, with AVX (4 doubles), hyper-threading 32 cores
- 64 Gb memory

Interconnection switch InfiniBand

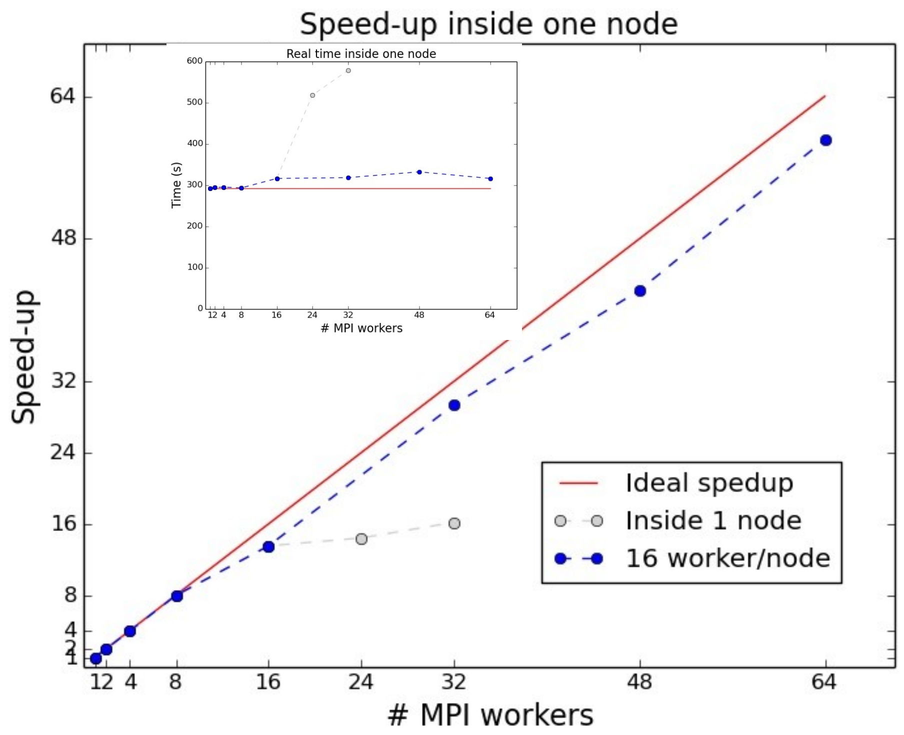
MPI Implementation Performance (1)



- Workload : 3 ev. per MPI worker, 20k points
- Deployment: ~ 1-2 worker(2) / core
- **Inside** 1 node: 16 physical cores, 32 virtual cores
- Speed-up:
  $g = t_P * N / t_{P=1}$
  with P : # of MPI workers

Hyper-threading leads to an efficiency loss !
Not general rule … depends on the application

MPI Implementation Performance (2)



- Workload : 3 ev. Per MPI worker, 20k point
- Deployment: ~ 1 worker / core
- Several nodes (here 4 nodes)
- Good scaling

Production code (200 k events)

Run currently on 200-400 cores on MPI platform (< 2 days with a reduced numerical scheme)

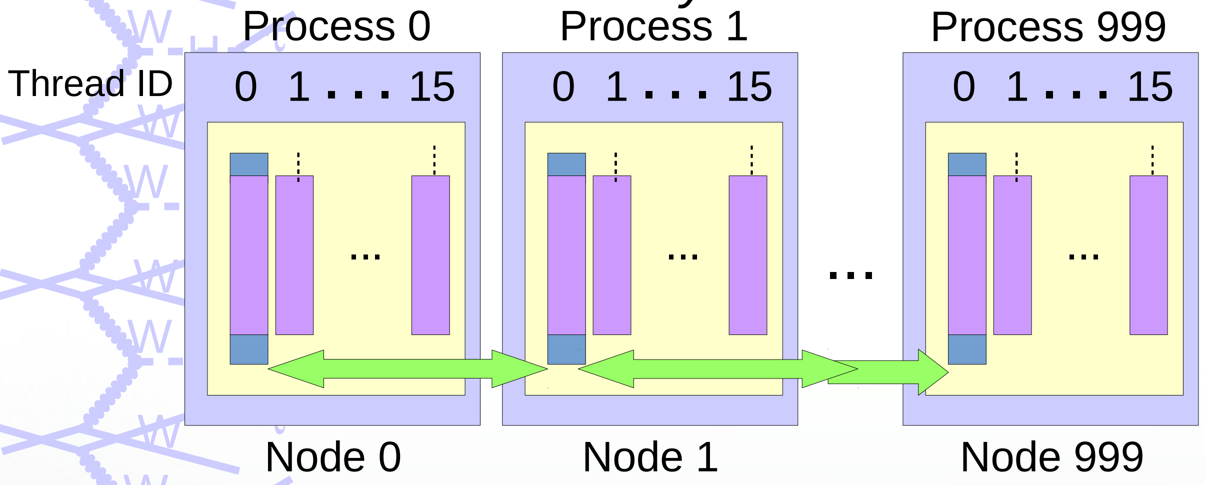
Efficient parallelism: good scaling & extensibility

MPI implementation Improvements

Open Multi-Processing (OpenMP or OMP). Standard widely used in HPC

- Share memory system (a node)
- Processing unit: a thread (MPI is a process)
- API with directives
`#pragma omp parallel for`

MPI/OMP hybrid model



Possible improvements:

- hybrid MPI/OpenMP: 1 MPI process per node and 1 thread per core (→ be replaced by OCL)
- Optimization (not the priority)
- // inside the integration part for high-dimensional integrals (σ_{tot})

MEM MPI is in production since the beginning of summer
Provided the analysis results for L. Mastrolorenzo thesis

HPC implementation

Accelerating the MEM analysis: OpenCL

- Technological possibilities:
  hardware, programming standards
- OpenCL concepts:
  OCL Abstract model
- Case study: reduction
- Conclusion

OpenCL
Technological choice (1)

- OpenCL users: embedded systems (smart phones, pad, …), large community
- HPC world : among the 10 most powerful supercomputers, 4 have accelerator cards (top500.org). Most of codes are developed with non standard programing paradigms (CUDA/kernels, OpenMP-like directives)
- OpenCL: not a big community in HPC word (CUDA) but more projects in HEP (GPU in HEP 2014, > 30 % talks with OCL, 2015 ~ 50% talks)

Programing paradigms:

- CUDA/kernels: NVidia (free)
- OpenACC/directives: NVidia (not free), CRAY (NVidia claims that is a standard !)
- OpenCL: AMD (free), Intel (free ?/Beignet), NVidia (free 1.1)
- Intel suite : TBB, Cilk, MKL
- OpenMP4: announced 2 years ago, recent commercial products, not free, close to OpenACC, Intel, (not NVidia/PGI why ?)
- Remark: Intel OpenCL free ? Intel Media Server Studio > $500 (provided with XeonPhi, beignet)

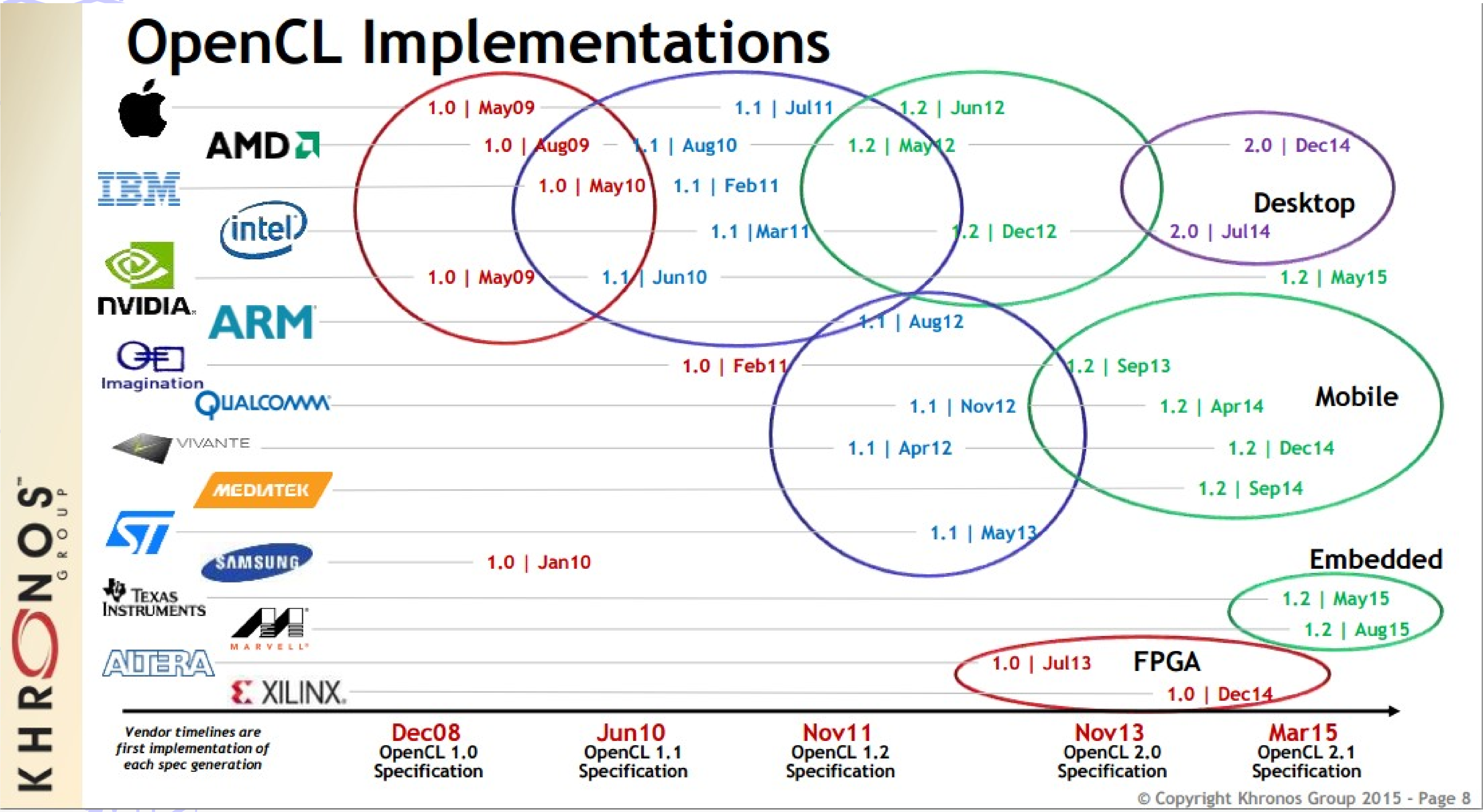
Hardware trends for computing:

- GPU on CPU chips: Intel (OpenCL/Windows), AMD
- CPUs + FPGA: Intel (with OpenCL)
- Real many-core processors: Intel Phi KNL

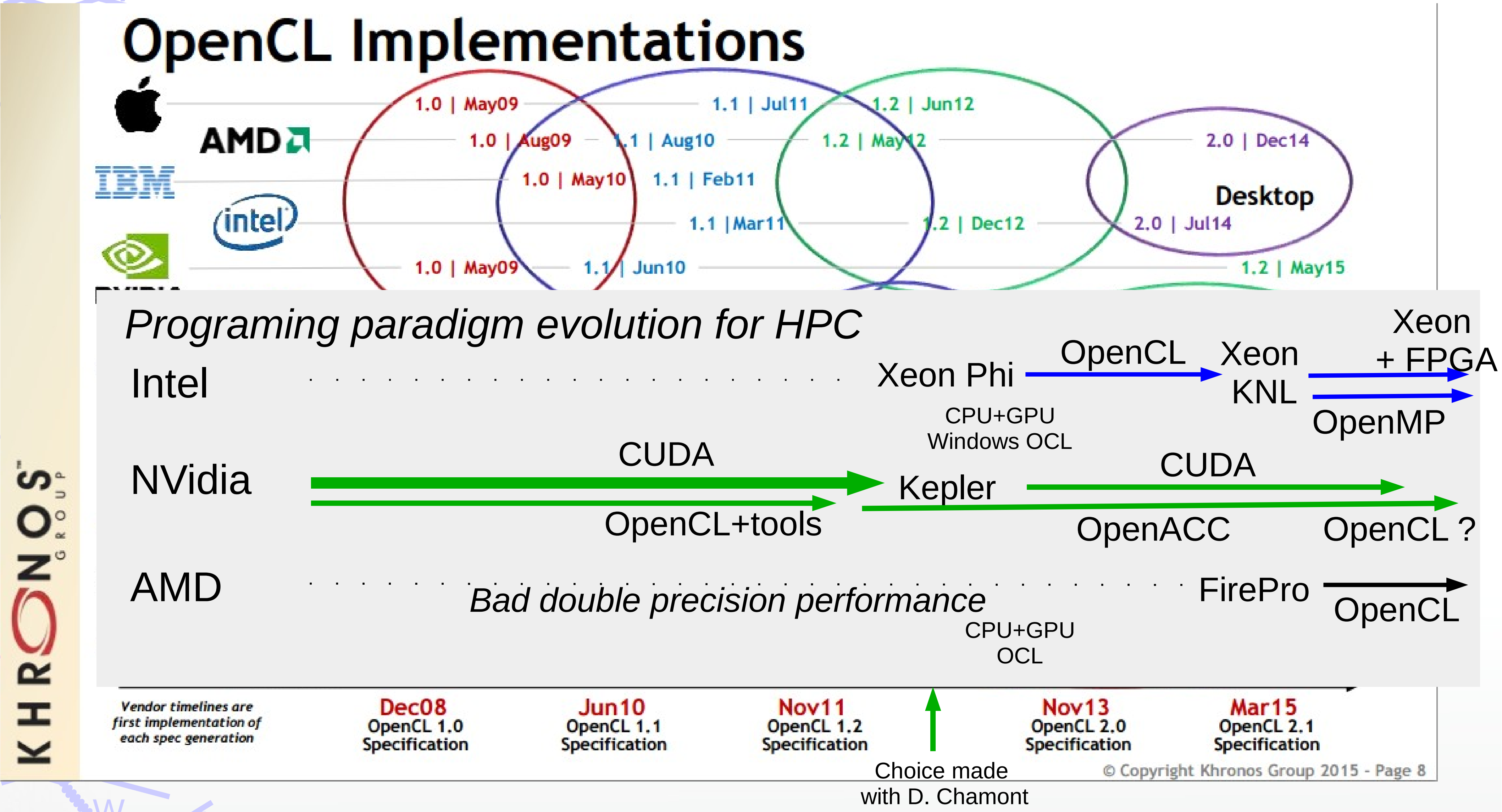
Commercial competition … OpenCL & OpenMP true standards

… OpenCL at technology hub

OpenCL
Technological choice (2)

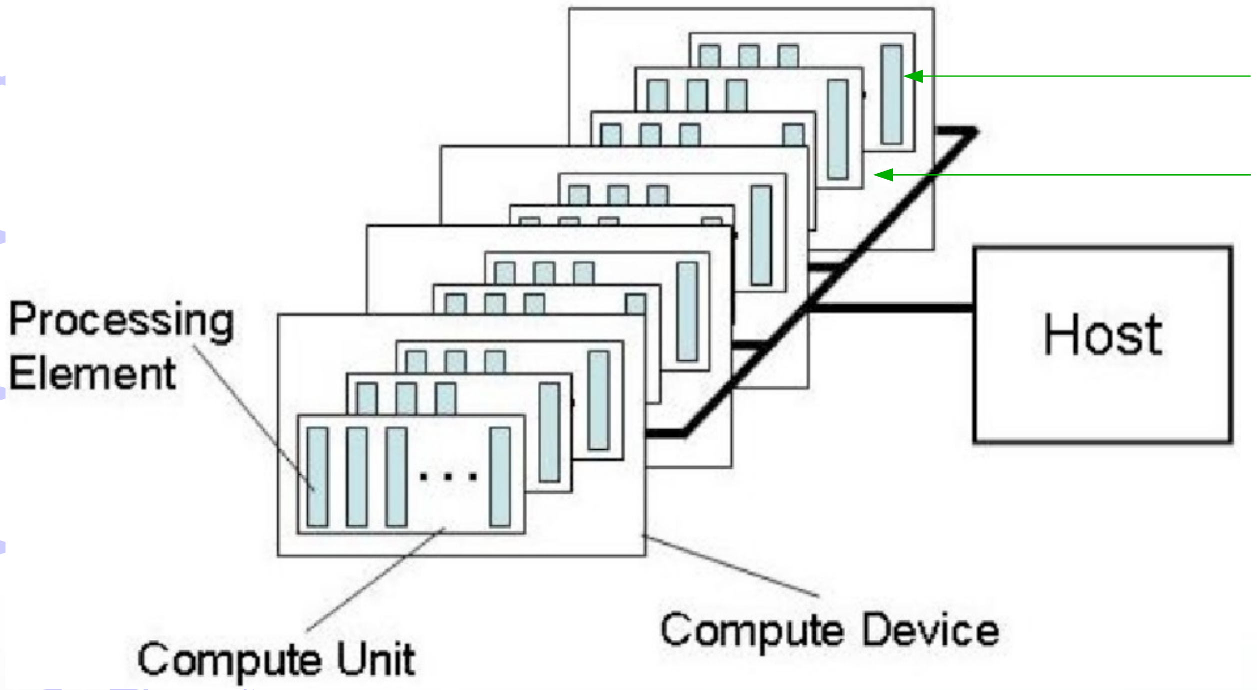


OpenCL
Technological choice (2)



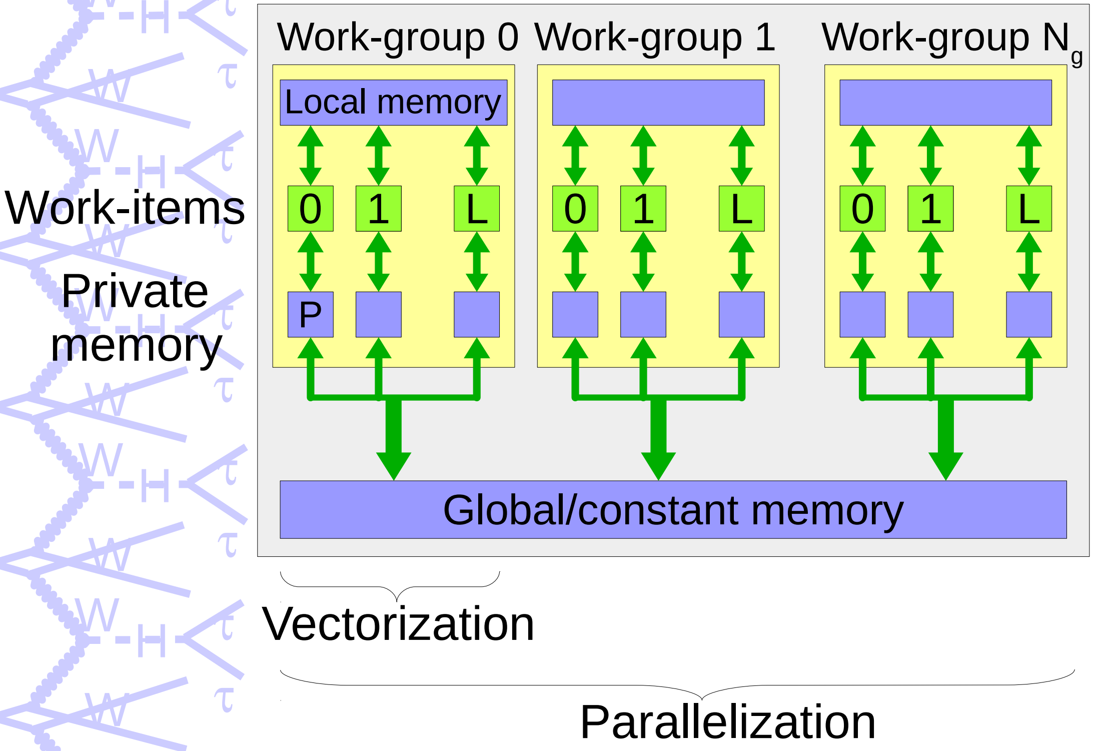
OpenCL Abstract Model (1)

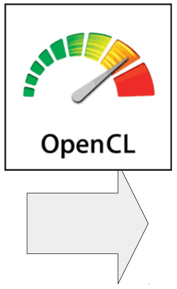
Open **Computing** Language (OCL)



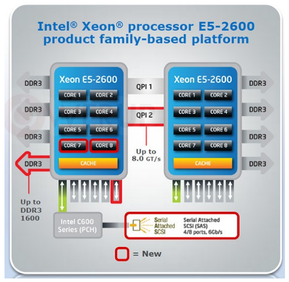
- Handle heterogeneous hardware
- Kernels (C99) → OCL devices
- Host/device dialogue (orders, data copies)
- Hybrid // model:
  - Shared memory model inside one device
  - Distributed memory model: host/device interactions, multi-devices programming

OpenCL
Abstract Model (2)



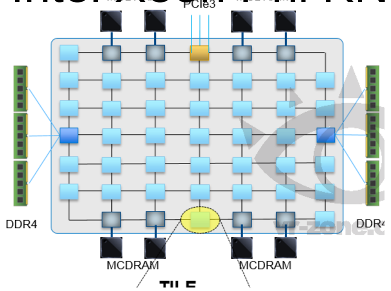


Intel CPU E5-2650

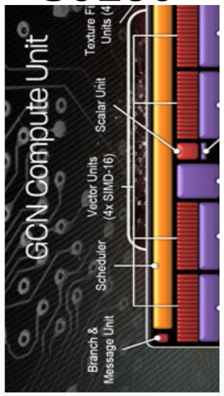


Nvidia Kepler K20

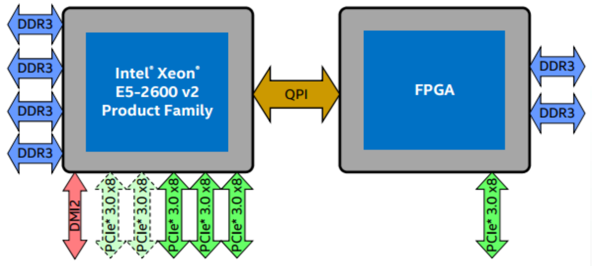
Intel Xeon Phi KNL



AMD FirePro S9150

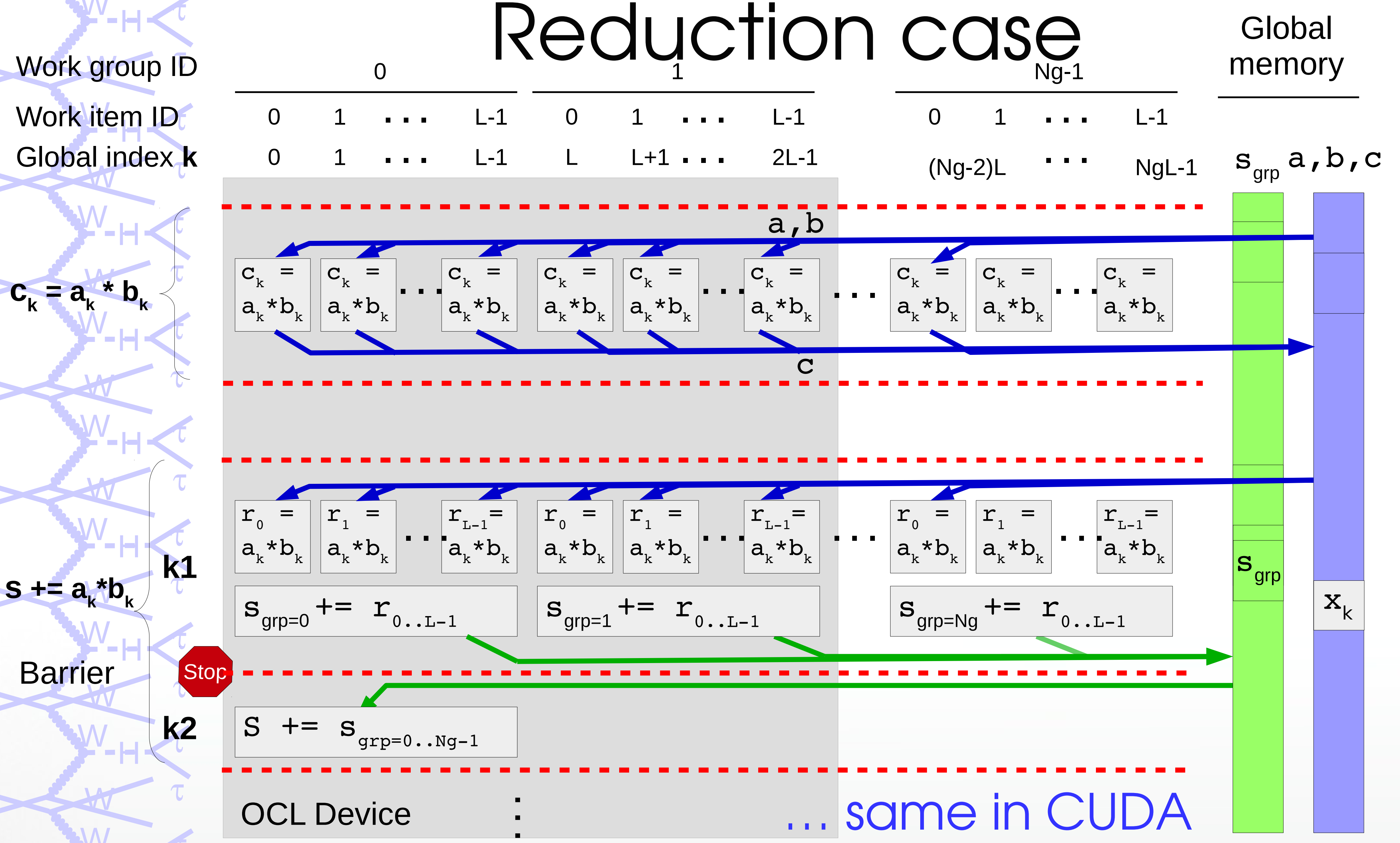


Intel Xeon + FPGA



OpenCL design to target vectorial & parallel hardware

OpenCL
Reduction case



OpenCL
Conclusion

- Kernel technologies (OpenCL, CUDA) are not simple
- Directive technologies (OpenACC, recent OpenMP) often lead to efficiency problems (→ code refactoring)
- C++ paradigms: vectors and Object Languages not friends → vector (arrays) low level, OOP at high level OK
- Rich CUDA environment (dbg, performance analyzer, libraries)
  Not the case for OCL but more and more libraries (ArrayFire, Linear Algebra, FFT, …)
- CUDA cannot support heterogeneous hardware even CPUs (CUDA + **OpenMP** → ok)
- OpenCL open standard widely used ← choice
- Fall-back solution:
  - Easy to change OpenCL → CUDA kernels
  - Built an interface `opencl.hpp` which switches OpenCL calls to CUDA calls give access to CUDA tool suite for OpenCL code … works well (clEvent to be integrated ...)

HPC implementation

MEM OpenCL

- MEM Engine description:
  VEGAS architecture in OCL
- Random Number Generator in OCL
- LHAPDF implementation in OCL
- MadGraph 5:
  OCL code generation
- How to distribute the work on several accelerator:
  DeviceDispatcher
- MEM Integration (ROOT, ...)
- Numerical validation
  Not trivial with different parallelism levels

OCL VEGAS
Context

Seems to be easy to // on GPUs but if we want an efficient implementation:

- Minimize the synchronization points (reductions)
- Keep the computation of the chi-square
- Take advantage all “many-core” accelerators available
- Take advantage all CPU-cores
- Hard to debug with OpenCL

… not so easy

OCL VEGAS
Brief Introdction

MC, M points $x \in V$

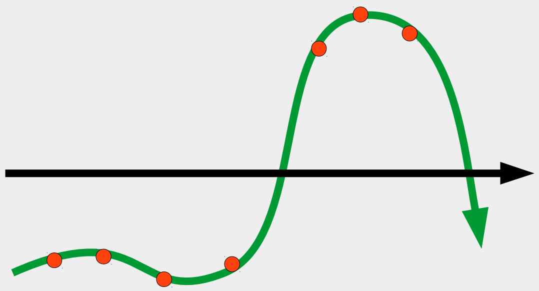
$$I \xrightarrow[M \to \infty]{} \frac{V}{M} \langle f(x) \rangle_p$$ with *p(x)* probability density

→ VEGAS algorithm, G.P. Lepage, J. Comput. Phys. 27 (1978) 192

- **Importance sampling**

$$p(x) \propto |f(x)|$$

→ *concentrate sampling to minimize* σ^2

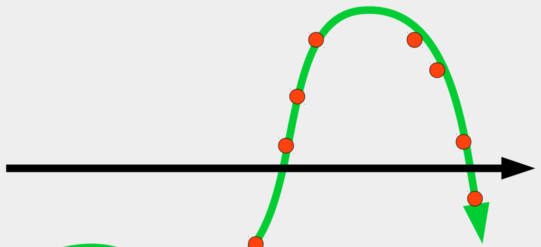


- **Stratified sampling**

V = U V$_i$ *with i = 1 .. N*

$$\sigma_i^2 = \sigma^2 / N$$

→ *concentrate sampling for high* σ^2
i.e. p(x) α *|df(x)/dx|*



OCL VEGAS
Parallelism with OCL (1)

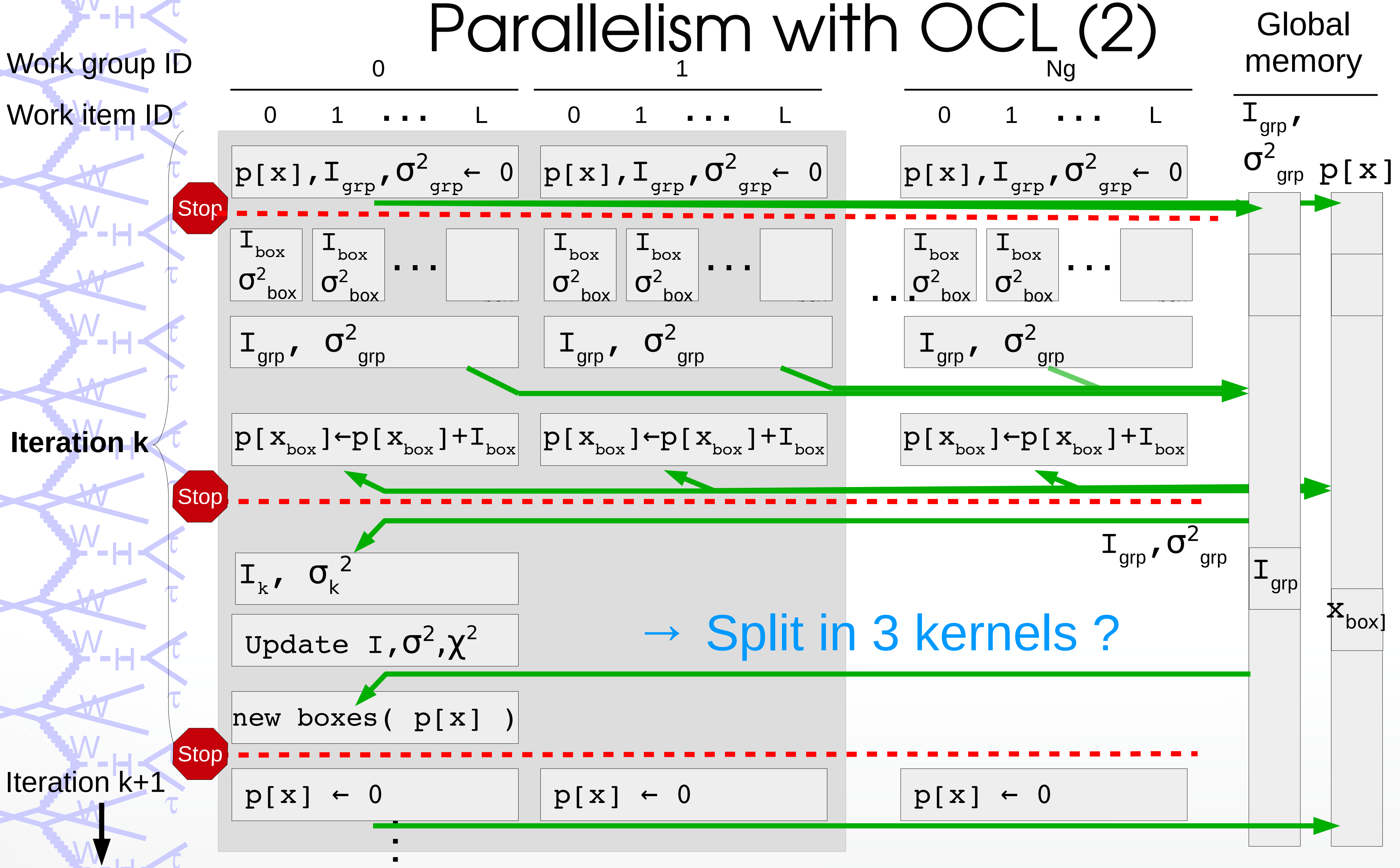
Simplified numerical scheme

```
Loop (k<5) <Chisquare>
 $I_k, \sigma^2_k(I)$, p(x) ← 0
W Loop <boxes>
   Loop <points per boxes>
    x = rand()
    $I_{box}(\mathbf{x})$ = vol(x) * F(x)
    update $I_k, \sigma^2_k(I)$, p(x)  Stop
   End Loop
  End Loop
  update $I, \sigma^2(I), \chi^2$
  Adjust boxe sizes ← p(x)
End Loop
```

- Workload to distribute: `<boxes>` loop
- Reduction: memory synchronization of $I_k, \sigma^2_k(I)$, p(x)
- External loop "`<Chisquare>`" troublesome
- Turned out to be not possible to avoid splitting:
  - Not possible to synchronize WorkGroups (CUDA)
  - No possibility to set the *Work Size* to the # *Computing Units*
  - No simultaneous Kernels with Nvidia OpenCL 1.1

Constrain to split the kernel !

OCL VEGAS
Parallelism with OCL (2)

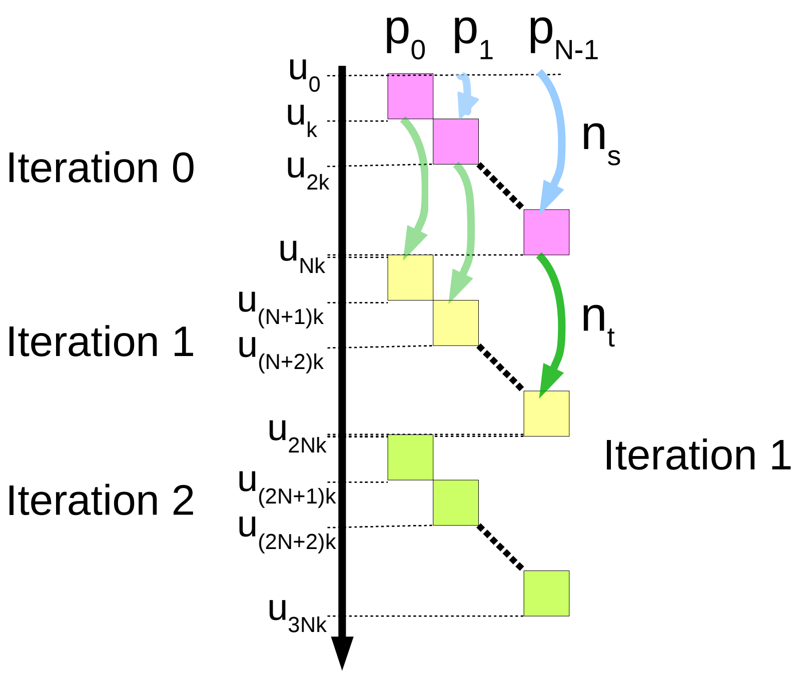


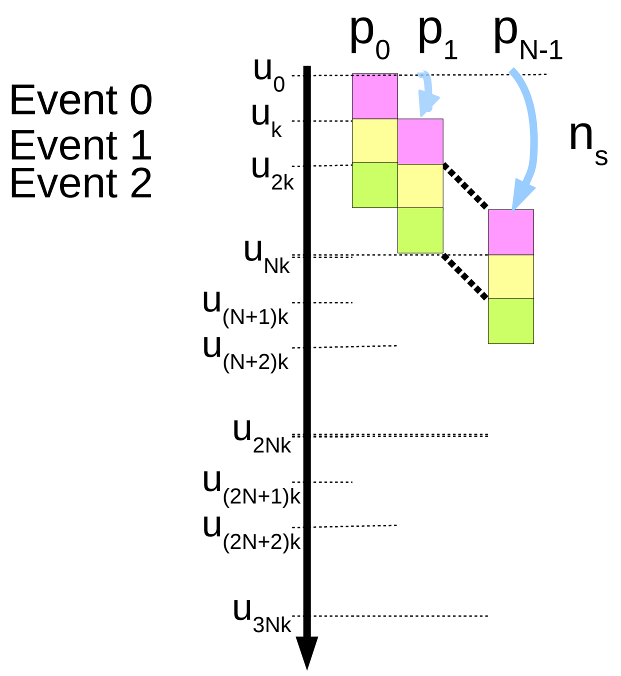
Global memory
Work group ID
0
1
Ng
Work item ID
0 1 ... L
0 1 ... L
0 1 ... L
I_{grp}, σ^2_{grp}
p[x]
$p[x], I_{grp}, \sigma^2_{grp} \leftarrow 0$
Stop
I_{box} σ^2_{box}
...
I_{grp}, σ^2_{grp}
Iteration k
$p[x_{box}] \leftarrow p[x_{box}] + I_{box}$
Stop
I_{grp}, σ^2_{grp}
I_{grp}
$x_{box}]$
I_k, σ_k^2
Update I, σ^2, χ^2
new boxes(p[x])
Stop
Iteration k+1
$p[x] \leftarrow 0$

→ Split in 3 kernels ?

OCL Random Number Generator
Parallelism context

Random sequence distributed among concurrent processes





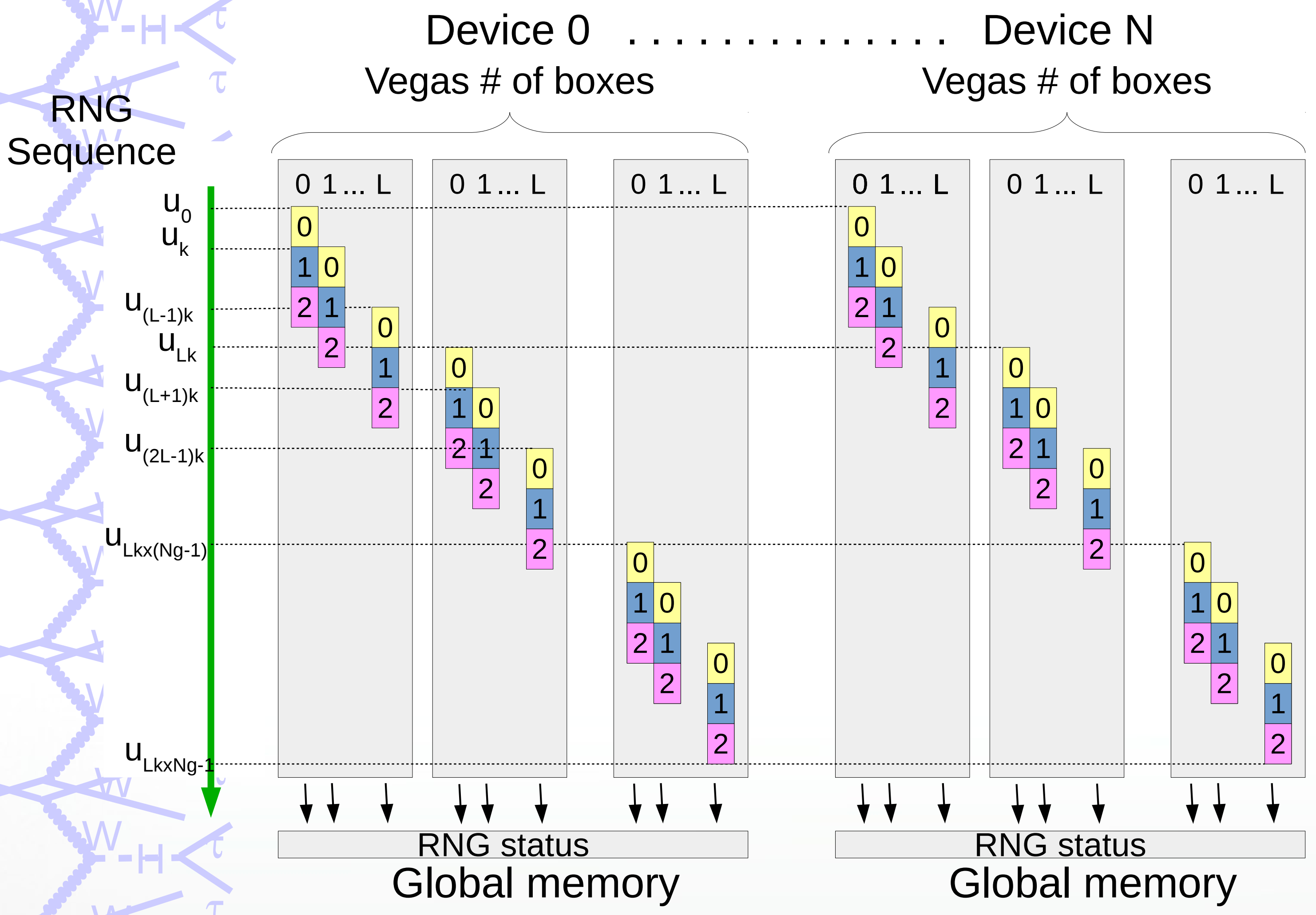
RNG & parallelism (classical)

- To avoid correlation, shifts are performed (n_s, n_t iterations).
- With n_t = k x N, N number of processes
- Very large $n_t \sim 10^4$ x 2d x 10^2 x $N_{integrals}$

Optimization:

- $n_t = 0$
- No correlation inside an event

OCL Random Number Generator Implementation

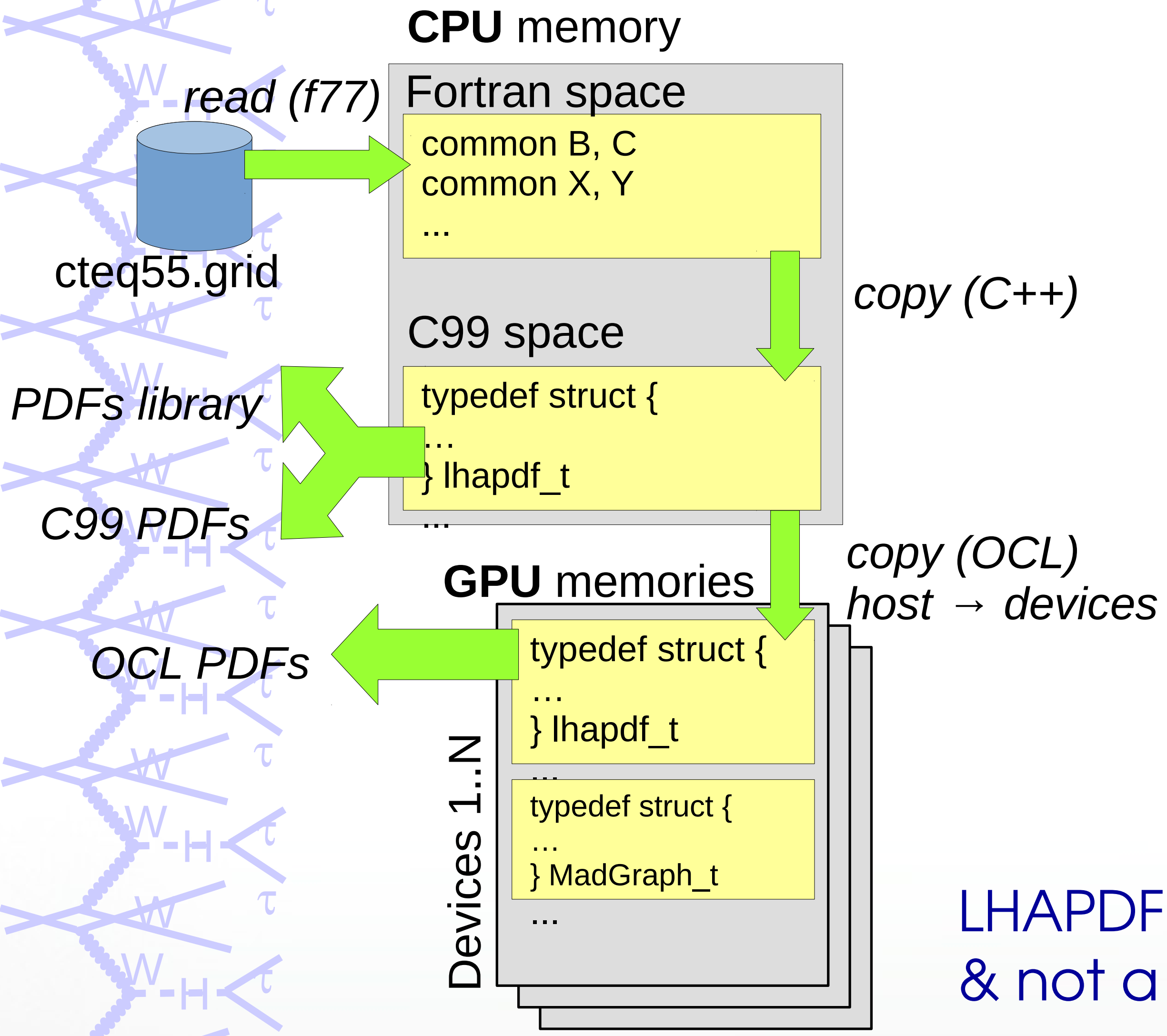


0 1 2 …events

L: Working group size
k: # of RN per box

- Use the GSL library (rand48)
- Global memory allocation per device: N_{vegas} points
- Debugging use reproducible cycles

OCL LHAPDF



- LHAPDF 5.8.5
- Sub-set CTEQ65d (used by a reference code L. Bianchini)
- Fortran → C99 (OpenCL)

  Deals with *global* variables

  Starting index of arrays (1 → 0)
- Validated numerically independently from MEM code with a large spectrum of parameters (Q, x, partons)

LHAPDF C++ 6.x.x … not the priority & not a "technological wall"

MadGraph
Features required

Extend MadGraph5_aMC@NLO to generate C99/OCL code (standalone_ocl)

Requirements (mandatory)

- Context Standard Model (SM)
- No *complex* type, no operators (present in CUDA)
- No *static* variable (make no sense in OCL)
- Kernel key-word extension: __kernel, __global, …

Features to facilitate OCL MG integration

- MG5 C++ classes → C99-struct
- One class (or struct) by (sub)process is not tractable → (sub)process array(s)
  Ex: Process_uu_h_uu.method() → process[uu_h_uu]
  Solves the copies host/devices pb
- One source file to check all processes (check_sa_ocl_all.cc)

Not a MadGraph specialist → maybe there are better ways to generate OCL code

MadGraph
OCL code generation

Universal FeynRules Output (UFO) models/sm:

- couplings.py (generated by Mathematica)

```
GC_81 = Coupling(
  name = 'GC_81',
  value = 'ee**2*complex(0,1)*vev
    + (cw**2*ee**2*complex(0,1)*vev)
      / (2.*sw**2)
    + (ee**2*complex(0,1)*sw**2*vev)
      / (2.*cw**2)',
  value_fct = 'Cmplx( 0, ee*ee*vev
    +  (cw*cw*ee*ee*vev) / (2.*sw*sw)
    + (ee*ee*sw*sw*vev) / (2.*cw*cw)  )',
  order = {'QED':1} )
```

- object_library.py: value_fct propagation:
  class Parameter(UFOBaseClass)
  class Coupling(UFOBaseClass)

ALOHA
Automatic Language-independent Output of Helicity Amplitudes
Generate code for amplitude computations from UFO (based on HELAS)

- *aloha/aloha_writers*.py

  → class WriteALOHA: reinterpret expression tree (complex)

  - variable type : double or complex
    typeOfVariable = {'F':'c', 'V':'c', 'T':'c','S':'c', 'M':'d', 'W':'d', 'P':'d'}
  - IsComplex(expression) for variables and simple expressions
  - getFunctionName(self, arg1, arg2, cmplx1, cmplx2, op):
    a + b → a + b
    z1 + z2 → CmplxAdd(z1, z2)
    a + z → CmplxAdd_dble(z, a) ⬌
  - write_MultVariable method, write_obj_Add, write_MultContainer

    propagate type: ex d, d → d; c, d → c; …
    call GetFunctionName()

  → New class ALOHAWriterForOCL(WriteALOHA)

- aloha/template_files

  → replace complex operator (iostv)xxxxx.cc with function syntax (HELAS):
  ixxxxx.c oxxxxx.c sxxxxx.c txxxxx.c vxxxxx.c
  xxxxxx_c.h (global header)

HELAS unchanged
HELicity Amplitude Subroutine library.

MadGraph
OCL code generation

MadGraph

- madgraph/interface/madgraph_interface.py:
  class MadGraphCmd.export ()
  Deals with new max variables for ninitial, nprocesses, nexternal, ncomb, namplitudes, nwavefuncs, ncolors → memory allocation.
- madgraph/iolibs/ufo_expression_parsers.py
  class UFOExpressionParserOCL(UFOExpressionParser)
  Parser for UFO algebraic expressions → OCL code
- madgraph/iolibs/helas_call_writers.py
  class OCLUFOHelasCallWriter(UFOHelasCallWriter)
  Generate the code to compute Helicity Amplitudes (based on HELAS)
- madgraph/iolibs/file_writers.py
  class OCLWriter(FileWriter):
  Format OCL code, keeps track of brackets, spaces, indentation and splitting of long lines
- *madgraph/iolibs/export_ocl.py*
  Organize the code generation of the (sub)processes (dir, make, files, template files)
- *madgraph/iolibs/template_files/OCLLibrary*
  External OCL libraries (GSL complex, gslmath, …, read_slha_c.cc)

- madgraph/iolibs/template_files
  Template files

  → All subprocesses:
  ocl_allprocess_head_h.inc, ocl_allprocess_h.inc, ocl_allprocess_tail_h.inc, ocl_allprocess_head_c.inc, ocl_allprocess_c.inc, ocl_allprocess_fct_declaration.inc, ocl_allprocess_function_definitions.inc

  → Single subprocess:
  ocl_process_h.inc ocl_process_c.inc
  ocl_process_matrix.inc ocl_process_class.inc (C struct)
  ocl_process_sigmaKin_function.inc
  ocl_process_function_definitions.inc
  ocl_process_wavefunctions.inc ocl_hel_amps_c.inc
  ocl_hel_amps_h.inc ocl_model_parameters_c.inc
  ocl_model_parameters_h.inc

  → Stand alone check
  - Single subprocess: check_sa_ocl.cc
  - All subprocesses: ocl_check_all_head_c.inc ocl_check_all_c.inc, ocl_check_all_tail_c.inc

  → Make files:
  - Single subprocess: Makefile_sa_ocl_src, Makefile_sa_ocl_sp
  - All subprocesses: Makefile_sa_ocl_All

MadGraph (4)
Code generation - Amplitudes

HelAmps_sm.cc

```
void FFV3_3( complex<dble> F1[],
 complex<dble> F2[], complex<dble> COUP,
 double M3, double W3, complex<dble>  V3[] ) {

 complex<dble> cI=complex<double>(0.,1.);
 complex<dble> TMP2, TMP0, denom;
 double P3[4], OM3;
  …
 TMP0 = (F1[2] * (F2[4] * (P3[0] + P3[3])
       + F2[5] * (P3[1] + cI * (P3[2])))




       + … );
 TMP2 = …
 denom = …
 V3[2] = …
 V3[4] = denom * - 2. * cI * (
  OM3 * 1./2. * P3[2]    * (
  + 2. * (TMP2) - TMP0) + (

  - 1./2. * cI * (F2[5] * F1[2])
  + 1./2. * cI * (F2[4] * F1[3])
          - cI * (F2[3] * F1[4])
          + cI * (F2[2] * F1[5])
  ));
 V3[5] = …
}
```

HelAmps_sm.c (OCL)

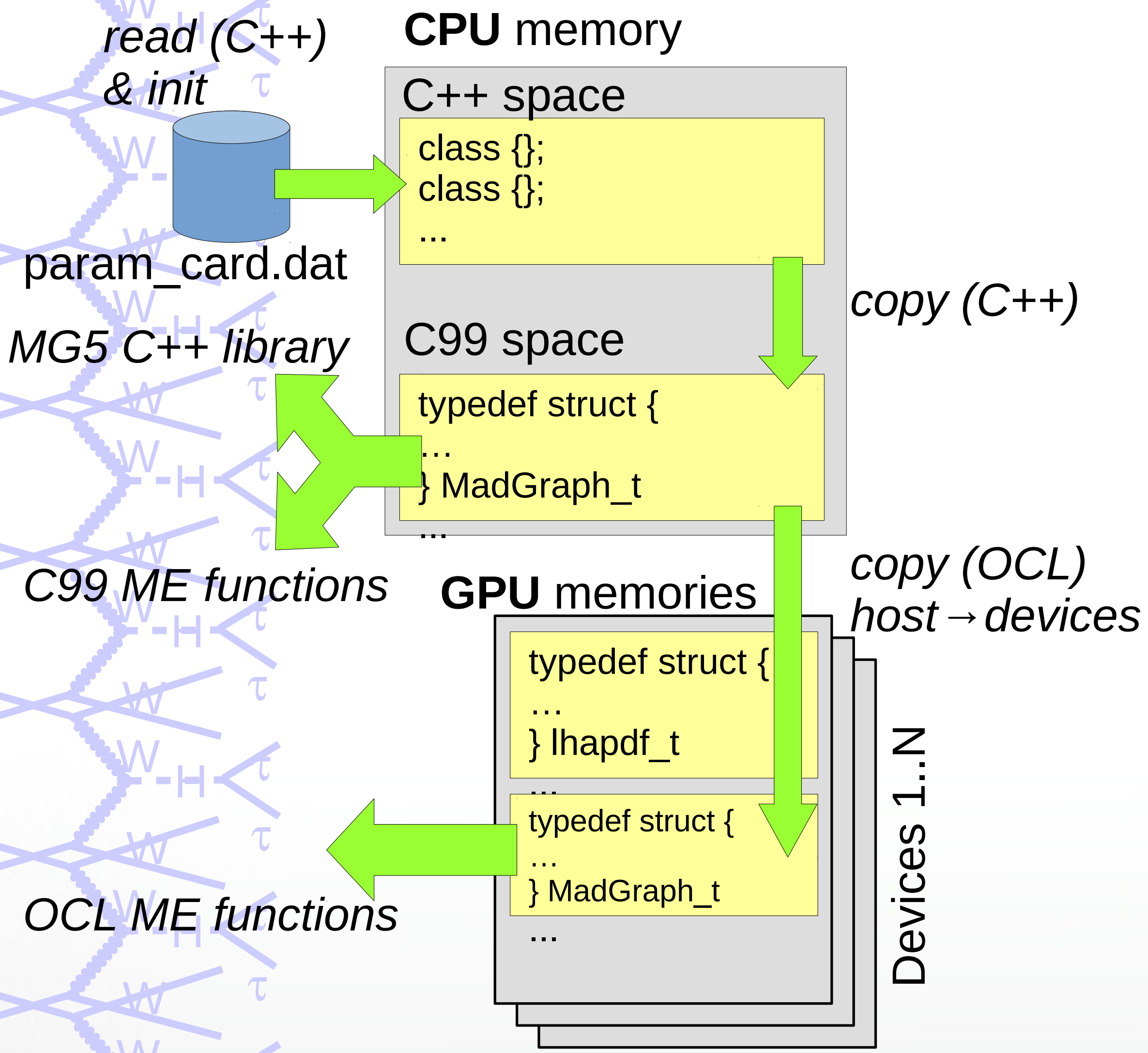
```
void FFV3_3( complex_t F1[], complex_t F2[], complex_t
        COUP, double M3, double W3, complex_t V3[]) {


 complex_t cI = {0., 1.}, cCI = {0., -1.};
 complex_t TMP2, TMP0, denom;
 double P3[4], OM3;
 …
 TMP0 = CmplxAdd(
    CmplxMul(F1[2],CmplxAdd( CmplxMul_dble( F2[4],
    (P3[0]) + (P3[3])), CmplxMul(F2[5],
    CmplxAdd_dble( CmplxMul_dble( cI, P3[2]),
    P3[1])))),
     … );
 TMP2 = ...
 denom = ...
 V3[2] = …
 V3[4] = CmplxMul(denom,CmplxMul(CmplxAdd(
  CmplxMul_dble(CmplxMul_dble(CmplxAdd(
  CmplxMul_dble(TMP0, -1.), CmplxMul_dble(TMP2, 2.)),
  (P3[2])*(1./2.)),OM3),CmplxAdd(CmplxAdd(CmplxAdd(
  CmplxMul(CmplxMul(F2[5],Cmplx(0, -1./2.)), F1[2]),
  CmplxMul(CmplxMul(F2[4],Cmplx(0,  1./2.)), F1[3])),
  CmplxMul(CmplxMul(F2[3],            cCI), F1[4])),
  CmplxMul(CmplxMul(F2[2],            cI ), F1[5]))),
  Cmplx(0,-2.)));
 V3[5] = …
}
```

MadGraph
Example: Check All

```
...
# include "OCLProcess.h"

const char * CardName = ".../param_card.dat";
OCLProcess_t oclProcess[ NbrOfProcesses ];
double oclME[ MaxNProcesses ];

int main(int argc, char * * argv) {
  // Get phase space point
  momentum_t p[6];
# include "input.h"

  // uu → h → uu Tau+ Tau-
  OCLProcess_initProc_uu_h_uutamtap(
        &oclProcess[ uu_h_uutamtap ], CardName );
  OCLProcess_getME_uu_h_uutamtap(
         &oclProcess[ uu_h_uutamtap ], p,
oclME );
  // Display
  for( i = 0; i < nprocesses_uu_h_uutamtap; i++){
    cout <<  "OCL ME= " << ...  <<  oclME[i] << …
         << -(2*nexternal_uu_h_uutamtap - 8) << …
  }

  // ud → h → ud Tau+ Tau-
  …
```

- `NbrOfProcesses` handle array of (sub)processes
- `MaxNProcesses` (permutations) all to allocate an array for all processes
- `uu_h_uutamtap` identifier for all the processes
- `nprocesses_uu_h_uutamtap` identifier for all the (sub)processes
- No need to handle `Parameters_sm_t` variable
- More generic functions : `OCLProcess_initProc()` `OCLProcess_getME()`

MadGraph Integration in MEM

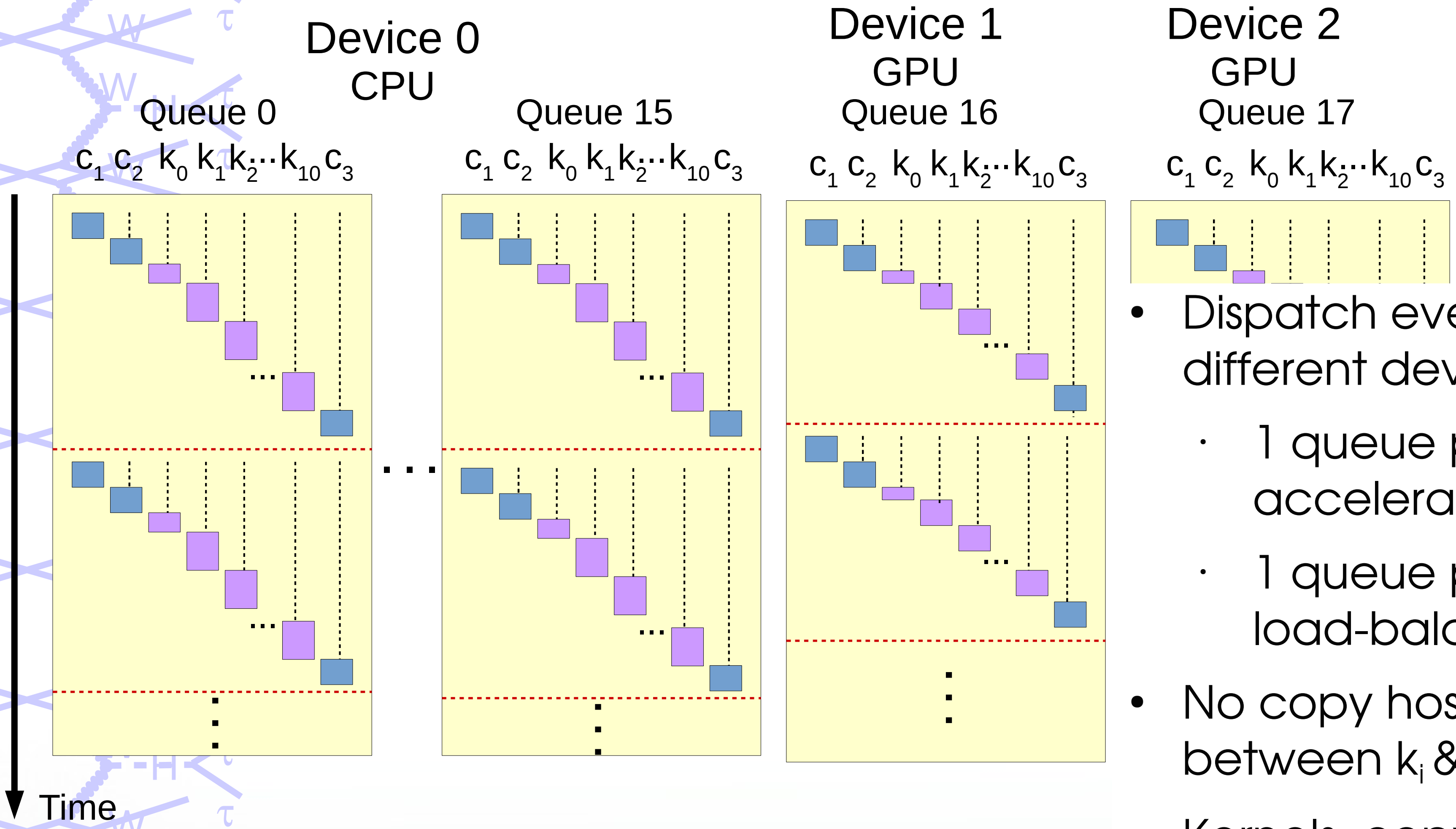


- Initialization with C++ library
- One `struct` per MG subprocesses
- Copies in C99 `struct`
- `MadGraph_t struct` simplify the copies to OCL devices

Numerical validation

- Check All
- Possibility to reproduce Thomas checks to reduce # of subprocesses
- Global validation (see later)

Device Dispatcher


Device 0
CPU
Queue 0
$c_1\ c_2\ k_0\ k_1 k_2 \cdots k_{10}\ c_3$
Queue 15
$c_1\ c_2\ k_0\ k_1 k_2 \cdots k_{10}\ c_3$
Device 1
GPU
Queue 16
$c_1\ c_2\ k_0\ k_1 k_2 \cdots k_{10}\ c_3$
Device 2
GPU
Queue 17
$c_1\ c_2\ k_0\ k_1 k_2 \cdots k_{10}\ c_3$
Time

- Dispatch events to different device queues:
  - 1 queue per accelerator
  - 1 queue per core (fine load-balancing)
- No copy host-device between k_i & k_{i+1} kernels
- Kernels, copy launched asynchronously

Asynchronous mechanisms are mandatory to manage several devices

Minimization of synchronization points & and tuning the # of cores

OCL Integrand

ROOT algebra & ME Implementation

	C++	**OCL kernel (C99)**

	`double MGIntegration::evalME(const double* x ) {`
	`...`
		`double evalME( double *x, size_t dim, __global void *ptr ) {`
		`// evLep4P, EvHadSys4P  evJet1_4P evJet2_4P, evRecoMET4P`
		`LzVector_t evLep4P = LzVector_constr_global(`
		`&ev->evLep4P  );`
		`...`

	`// Building frames`
	`TVector3  e_lep = Lep3P.Unit();`
		`// Building frames`
		`Vector3_t  e_lep = Vector3_unit( &Lep3P );`

Building leptonic Tau 4-momentum	`...`
	`// 1st frame (e_lep, e_x, e_y)`
	`TVector3 taulep = P_TLep * e_lep;`
	`double E_TLep = Sqrt( P_TLep*P_TLep + Physics::mTau2 );`
	`TLzVector TLep4P(taulep, E_Tlep);`
		`...`
		`// 1st ref. sys. (e_lep, e_x, e_y)`
		`Vector3_t taulep = Vector3_mult_double( &e_lep, P_TauLep);`
		`double E_TauLep = sqrt(P_TauLep*P_TauLep +Physics_mTau2);`
		`LzVector_t TauLep4P = LzVector_constr( &taulep, E_TauLep);`

Building hadronic Tau 4-momentum	`...`
	`double P_THad = getPTHad( m_TauTau_2, P_TLep, ... );`
	`double cosThTLepPi = TMath::Cos( taulep.Angle(e_pi) );`
	`...`
	`TVector3 tauhad = P_THad * (alpha*e_tau + beta*e_pi`
	`+ gamma*e_yy);`
		`...`
		`double P_THad = getPTHad( m_TauTau_2, P_TauLep,  ... );`
		`double cosThTauLepPi = cos( Vector3_angle(&taulep, &e_pi));`
		`...`
		`double a = alpha*P_THad,  b = beta *P_THad,  c = gamma  ...`
		`Vector3_t tauhad = Vector3_sum_3Vector3(a, &e_tau, b,`
		`&e_pi,  c,  &e_yy );`

Final State Quarks	`...`
	`//  Get the total reconstructed 4-momentum (5 terms)`
	`rho4P = evMET4P_+ evHadSys4P_+ evLep4P_ +  ...`
		`...`
		`// Get the total reconstructed 4-momentum`
		`rho4P = LzVector_sum_5(  ...  );`

Transfer fcts, weighted ME	`...`
	`//  JET, MET, ... Transfer functions`
	`...`
		`...`
		`// JET Transfer functions  More efficient, no histogram built`

Class

Method

Fct overload

Not big changes, but how to maintain two versions ?

Numerical validations

Sequential & parallel implementations

- Numerical validation of the different components (separately): LHAPDF, VEGAS, MadGraph
- Entire computation: VEGAS ↔ Regular Grid

Specific for parallel implementation

- Reproducibility: set the same RNG sequence (to be done for OCL)
- RNG ← 2-points cycle (OCL)
- VEGAS ↔ Regular Grid

Large spectrum of numerical checks

HPC implementation

Performance

- Different kind of hardware tested, including CPUs
- Conclusion – Performance synthesis

Performance

CPUs

25 000 lines of kernel code compiled for K20 GPGPU and/or Xeon Phi

- Benchmark (1 ME alone): uu → h → uu tau tau
- Hardware: 2 x E5-2650 16 cores, with AVX (4 doubles)
- Reference: exec. time on 1 core, for 1 event (s/event)

- Performance on CPUs

	GSL 1 core	MPI 16 cores	OCL 16 cores
Time/ev (s)	24.7	1.69	0.53
Speed-up	1	14.6	46.6

x 3.2

The important speedup comes from Intel OpenCL implementation which takes advantage of CPU vectorization features (AVX)

Performance
NVidia K20

- Performance on NVidia K20 – reference on **1-core**

	GSL 1-core	OCL 16-cores	OCL 1 x K20	2 x MPI/OCL 2 x K20	2 x MPI/OCL 2 x K20 + 16 c
Time/ev (s)	24.7	0.53	0.32	0.18	0.16
Speed-up	1	46.6	77.1	140.5	155.0

- Performance on NVidia K20 – reference on **1-node**

	MPI 16-cores	OCL 16-cores	OCL 1 x K20	2 x MPI/OCL 2 x K20	2 x MPI/OCL 2 x K20 + 16 c
Time/ev (s)	1.69	0.53	0.32	0.18	0.16
Speed-up	1	3.19	5.28	9.61	10.60

CHEP 2015 “Matrix Element Method for High Performance Computing platforms”
13-17 April, Okinawa

Performance
Intel Xeon Phi

- Reference on 1-core

	GSL 1-core	OCL 16-cores	OCL 1 x Phi	OCL 2 x Phi	OCL 2 x Phi + 16 c
Time/ev (s)	24.7	0,53	1,31	0,63	0,33
Speed-up	1	46,6	18,9	38,9	74,2

- Reference on 1-node

	MPI 16-cores	OCL 16-cores	OCL 1 x Phi	OCL 2 x Phi	OCL 2 x Phi + 16 c
Time/ev (s)	1.69	0.53	1.31	0.63	0.33
Speed-up	1	3.19	1.29	2.66	5.08

CHEP 2015 “Matrix Element Method for High Performance Computing platforms”
13-17 April, Okinawa

Performance
Conclusion

- 1 MG subprocess is a representative sample
- Our objective:
  1 node → 1 node + 2xK20s (2013 hardware) will accelerate by 10
- Probably more … to optimize

Cost study:

- Reference year for price & hardware performance: 2013
- For the same price
  20 MPI nodes (160 k€)
  replaced by
  11 nodes + 22 K20s

And …

… we should have results 5.5 time faster

… reduce cost of the energy supply (cooling)

HPC implementation

Conclusion

Future plan / Perspectives

- Short term tasks
- Investigate technologies:
  high level descriptions & performing automatic operations
- Final conclusion

Planned Tasks
Short term

Short term (Q4 2015)

- Deliver a MPI-OCL MEM benchmark to “evaluate new HPC architectures” (IBM/NVidia, Intel/KNL). A GENCI* project
- Finish the GPU implementation of Hττ channel. Adding background hypothesis (DY) Should be fast.

Short term (Q1 2016)

- Validation: production on GPUs (VBF/DY). Compare with MPI MEM results.
- Adding W+jets background
- Integrating in CMSSW (sequential in a first time)
- Refactoring: OCL dispatcher, ...
- Automatic tests with ≠ configurations

Short term (Q2 2016)

- ttH(→ττ) Analysis on GPU
- Run 2 (only I/O)
- Workload optimization (double MPI msg buffers)

***GENCI**: National Agency with drive HPC investments in France (→ Europe HPC agency) http://www.genci.fr

High-priority Investigations (1)

MEM prod. on GPUs (GridCL)

- $H\tau\tau$ → OCL/GPUs
- $ttH(\rightarrow\tau\tau)$ → OCL/GPUs

CMSSW MEM

- Integrate MEM in CMSSW: sequential (// MPI, OCL on CPU, OCL on GPU)

High dimension integrals

- Computation of σ_{tot} ($H\tau\tau$ dimension ~ 10) → distribution of the integral on several devices, MPI nodes

MEM factory

Exploding combinatorics with :

- SA Sequencial, //, OCL, OCL+MPI
- Provide modules for CMSSW, CMSSW+MPI+....
- Run 1, Run 2, …, MC
- Hypothesis (backgrounds) to test
- MadGrah code generation
- Integration mode Regular Grid, VEGAS, ...
- Automatic checks

→ aims to do a Factory to generate different kind of configurations

High-priority Investigations (2)

Towards Domain Specific Language (DSL)

- Recent talks on DSL:
  - French Conference on HPC (ORAP): lot of DSL experiments on super-computers
  - DataScience@LHC: talks on code generation, DSL for MEM (and NN)
- Pb 1: part of the code is cumbersome to write & very error prone (time lost to debug)
- Pb 2: have to manage several implementations of MEM: OCL, C++ … dangerous
- Pb 3: bad extensibility
- Idea: having a more abstract description of the ME computations with a specific language
- MadGraph is or is closed to have a DSL (UFO+ALOHA)
- First, reuse the MadGraph mechanisms (Python) to generate "error prone part of MEM" … then extend it

```
//@ define OCLProcess_getME getME
//@ define  matrix_elements Mes

// uu->uuH, cc->ccH, ubarubar->ubarubarH, cbarcbar->cbarcbarH
getME_uu_h_uutamtap( &process[uu_h_uutamtap], p0, MEs );
pdf_sum     = fu0*fu1+fc0*fc1+fubar0*fubar1+fcbar0*fcbar1;
ME2 += pdf_sum * matrix_elements[0];

// ud->udH, cs->csH, ubardbar->ubardbarH, cbarsbar->cbarsbarH
pdf_sum     = fu0*fd1+fc0*fs1+fubar0*fdbar1+fcbar0*fsbar1;
getME_ud_h_udtamtap(&process[ud_h_udtamtap], p0, MEs );
ME2       += pdf_sum * MEs[0];
//
getME_ud_h_udtamtap( &process[ud_h_udtamtap], p1, MEs );
pdf_sum     = fu1*fd0+fc1*fs0+fubar1*fdbar0+fcbar1*fsbar0;
ME2       += pdf_sum * MEs[0];
 //
getME_ud_h_udtamtap( &process[ud_h_udtamtap], p2, MEs );
pdf_sum     = fu0*fd1+fc0*fs1+fubar0*fdbar1+fcbar0*fsbar1;
ME2       += pdf_sum * MEs[0];
//
getME_ud_h_udtamtap( &process[ud_h_udtamtap], p3, MEs );
pdf_sum     = fu1*fd0+fc1*fs0+fubar1*fdbar0+fcbar1*fsbar0;
ME2       += pdf_sum*MEs[0];

// uc->ucH, ubarcbar->ubarcbarH
…
```

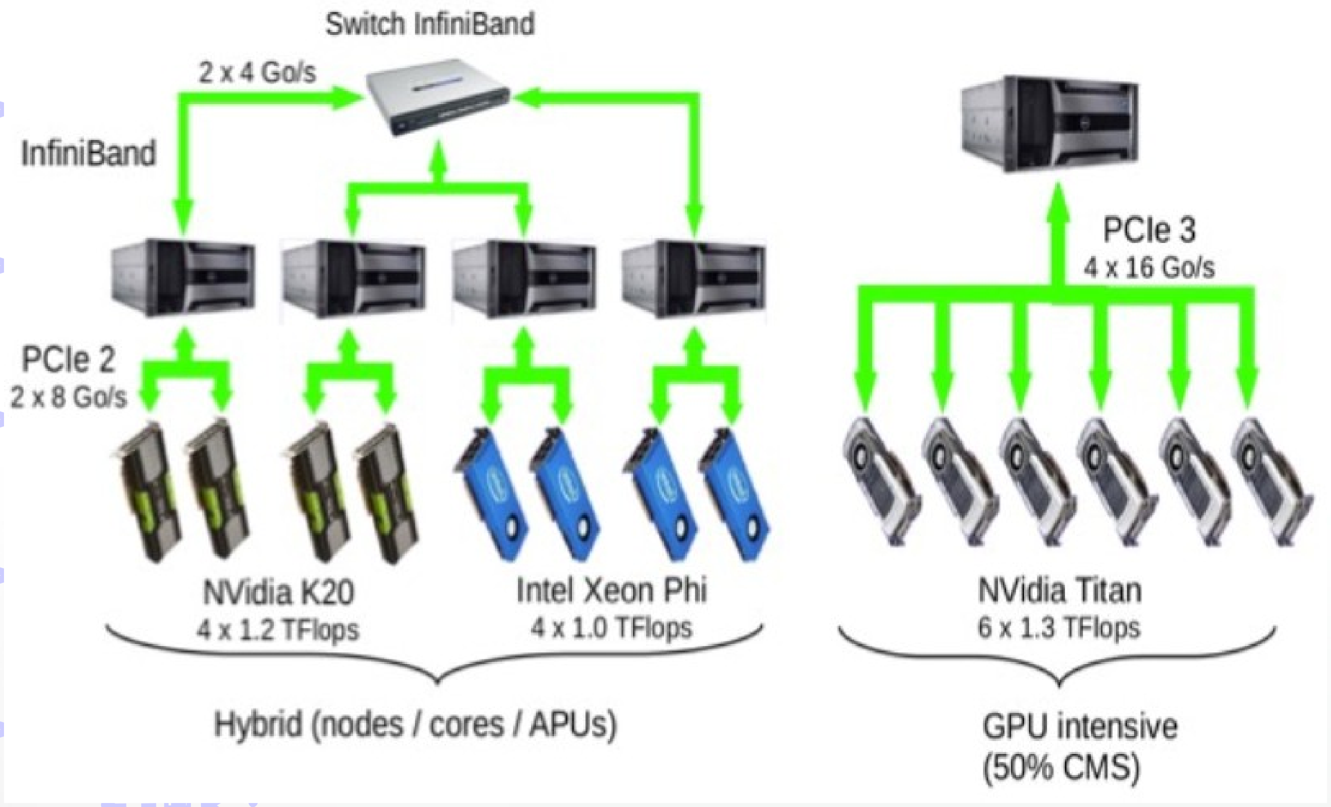
62 calls to OCLProcess_getMExytamtap (permutations) - 17 subprocesses involved for VBF

Conclusion

- MPI MEM producing analysis results for Hττ channel
- Improving the Hττ analysis by adding W+jets background
- Starting ttH(→ττ) analysis
- MEM analysis for Run 2 will require huge computing power
- Close to satisfy this request with OCL-MPI-MEM code
- OCL investment will be used for other projects

Thanks

- MEM Workshop organizers
- MEM-CMS (LLR) team:
  - MEM $H\tau\tau$: F. Beaudette, O. Davignon & L. Mastrolorenzo
  - MEM ttH: P. Paganini & T. Strebler



- IT Team: D. Chamont
- P2IO Funds for the GridCL platform