MCBooster: a tool for MC generation for massively parallel platforms

A. Augusto Alves Jr [1], Michael D. Sokoloff [1]

[1] University of Cincinnati, with support from NSF award PHY-1414736.

What is it?

MCBooster is a collection of templates (a header-only library) that generates large samples of phase space Monte Carlo and also can evaluate functions over these samples in parallel using CUDA and OpenMP enabled devices.

Phase space Monte Carlo: generalities

Describes the kinematics of the decay of a particle with a given four-momentum to an N-particle final state.

- No intermediate resonance between the final state particles.
- The number of independent parameters is $4N - 4 - 3 - N$.
- Uses the Raubold and Lynch method. [F. James, Monte Carlo Phase Space, CERN 68-15(1968)] [E. Byckling, K. Kajantie et al, Particle Kinematics]
- Well understood method: implemented in GENBOD (in CERNLIB) and TGenPhaseSpace (in ROOT)
- Each event generated is kinematically consistent.
- All generated events are accepted, and a weight is calculated for each to produce phase space distributions.

Implementation highlights

The library is implemented on top of the Thrust library and runs on Linux systems. MCBooster supports systems compatible with NVIDIA CUDA and OpenMP.

- Events are generated in large bunches (typically multiples of a million), depending on the memory available on the device.
- The complete set of four-vectors is available for the final states. MCBooster can output weighted and unweigted samples.
- Interfaces exist for intrusive and non-intrusive evaluation of arbitrary functions of the phase space coordinates.
- The MCBooster interface also supports generation of sequential decays.

class PhaseSpace and class Events

- Generation is managed by the class:

```
MCBooster::PhaseSpace(GReal_t _MotherMass, vector<GReal_t> _Masses,
                      GLong_t _NEvents)
```

- Generated events are kept on the device. The containers have the lifetime of the corresponding MCBooster::PhaseSpace object.
- Allocated memory can be reused in subsequent runs.
- The user can export generated events from the device to the host and store them in a container:

```
MCBooster::Events(GInt_t ndaughters, GLong_t nevents)
```

Function evaluation

- Three template functions cover most possibilities.

```
//single variable evaluation of arbitrary type
template<typename CUSTOMFUNC, typename RESULT>
void Evaluate(const CUSTOMFUNC funcObj, ParticlesSet_d &pset
, mc_host_vector<RESULT> &eval)

//more adequate for intrusive evaluation
template<typename CUSTOMFUNC>
void Evaluate(const CUSTOMFUNC funcObj, ParticlesSet_d &pset)

//evaluate array of variables at once
template<typename CUSTOMFUNC>
void EvaluateArray(const CUSTOMFUNC funcObj, ParticlesSet_d &pset,
VariableSet_h &varset )
```

- The user derives a class from the corresponding interface and implements the corresponding operator().

A worked example: $B^0 \to K^+\pi^- J/\psi$, with $J/\psi \to \mu^+\mu^-$

- The following example shows how to generate 10 million events of the decay chain and calculate the parameters corresponding to the $B^0 \to K^* J/\psi$, with $K^* \to K^+\pi^-$; $J/\psi \to \mu^+\mu^-$ which are:
  - $M(K,\pi)$, the invariant mass of the $K\pi$ system.
  - $M(J/\psi,\pi)$, the invariant mass of the $J/\psi\,\pi$ system.
  - $cos(\theta_K)$, the hecility angle of the K^*
  - $cos(\theta_\mu)$, the hecility angle of the J/ψ
  - $\Delta\phi$ difference between the decay planes of the K^* and J/ψ

$B^0 \to K^+\pi^- J/\psi$

```
// some input
GLong_t events = 10000000; //10 Million events
GReal_t massB0 = 5.2795;

//masses of B0 daugthers {J/psi,kaon , pion}
vector<GReal_t> massesB0={3.096916,0.493677, 0.13957018)};

//ctor
PhaseSpace phsp( massB0, massesB0, events);

//actual generation (all decays from same mother at rest)
phsp.Generate( Vector4R(massB0, 0.0, 0.0, 0.0) );

//event container
Events *MyEvents = new Events(3, events );

//export events to host
phsp.Export(MyEvents);
```

$J/\psi \to \mu^+\mu^-$

```
//J/psi daugthers {mu+, mu-}
vector<GReal_t> massesJpsi={0.100, 0.100};

//ctor
PhaseSpace phspJpsi( 3.096916, massesJpsi, events);

//see that I am passing the J/psi's from the B0 decay
//the generate method understand how many decay to
//generate, to where to boost etc
phspJpsi.Generate( phsp.GetDaughters(0) );

Events *MyEventsJpsi = new Events(2, events );
phspJpsi.Export(MyEventsJpsi);
```

Dataset calculation: functor

```
struct Dataset: public IFunctionArray
{
Dataset(){dim =4;}

__host__ __device__
GReal_t cosHELANG(Vector4R p, Vector4R q, Vector4R d )
{ //calculation }

__host__ __device__
GReal_t deltaAngle(const Vector4R& p4_p,
const Vector4R& p4_d1, const Vector4R& p4_d2,
const Vector4R& p4_h1, const Vector4R& p4_h2 )
{//calculation }

__host__ __device__
void operator()(const GInt_t n, Vector4R** particles,
GReal_t* variables )
{
Vector4R pJpsi = *particles[0];
Vector4R pK    = *particles[1];
Vector4R ppi   = *particles[2];
Vector4R pMup  = *particles[3];
Vector4R pMum  = *particles[4];

Vector4R pB0   =  pJpsi + pK +ppi; Vector4R pKpi  =  pK    + ppi;

variables[0] = pKpi.mass();
variables[1] = cosHELANG( pB0, pKpi, pK );
variables[2] = cosHELANG( pB0, pJpsi, pMup ) ;
variables[3] = deltaAngle( pB0, pK, ppi , pMup, pMum ) ;
}

};
```

Dataset calculation: evaluation

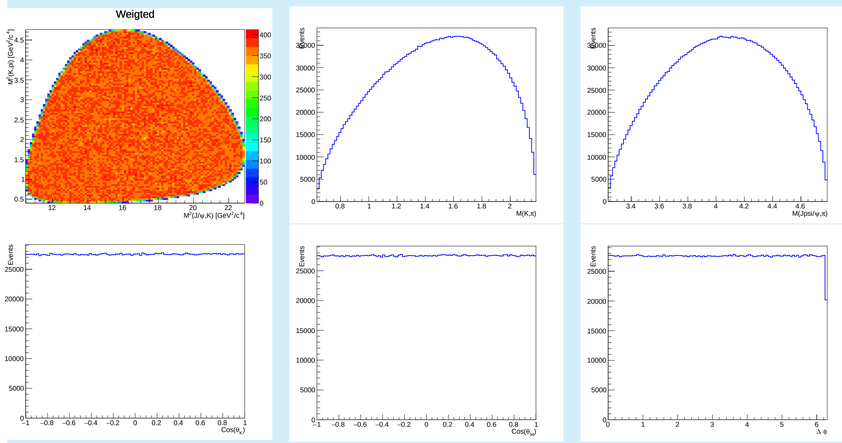
```
VariableSet_h Var(4);
RealVector_h result_MKpi(events);
RealVector_h result_CosThetaK(events);
RealVector_h result_CosThetaMu(events);
RealVector_h result_DeltaAngle(events);

Var[0]  = &result_MKpi;
Var[1]  = &result_CosThetaK;
Var[2]  = &result_CosThetaMu;
Var[3]  = &result_DeltaAngle;

ParticlesSet_d JpsiKpiMuMu(5);
JpsiKpiMuMu[0]   = &phsp.GetDaughters(0);
JpsiKpiMuMu[1]   = &phsp.GetDaughters(1);
JpsiKpiMuMu[2]   = &phsp.GetDaughters(2);
JpsiKpiMuMu[3]   = &phspJpsi.GetDaughters(0);
JpsiKpiMuMu[4]   = &phspJpsi.GetDaughters(1);

Dataset DataJpsiKpi = Dataset();
EvaluateArray<Dataset>(DataJpsiKpi, JpsiKpiMuMu, Var );
```

Results



Performance: CUDA

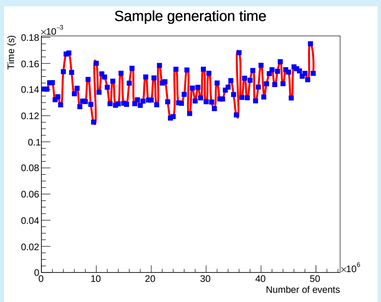
The time spent by different devices to process the 10 million events sample used in the previous exercise:

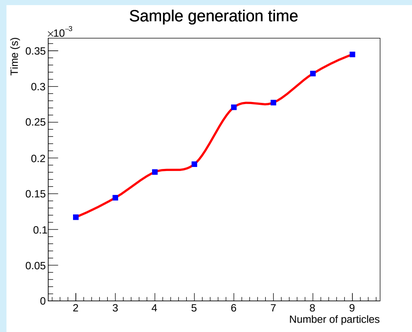
	Time spent per task for different cards (s)			
	K2200	GTX 970	GTX TITAN Z	K40c
$B^0 \to K\pi^+ J\psi$	0.034	0.016	0.020	0.021
$J\psi \to \mu^+\mu^-$	1.2×10^{-05}	8.78×10^{-06}	9.45×10^{-06}	1.41×10^{-05}
Dataset	0.18	0.10	0.078	0.049

- Quadro K2200 (Maxwell/5.0): 640 CUDA Cores @ 1.12 GHz. FP64/FP32 = 1:32
- GeForce GTX 970 (Maxwell/5.2): 1664 CUDA Cores @ 1.18 GHz FP64/FP32 = 1:32
- GeForce Titan Z (Kepler/3.5): 2880 CUDA Cores @ 0.88 GHz. FP64/FP32 = 1:3
- Tesla K40c (Kepler/3.5): 2880 CUDA Cores @ 0.75 GHz. FP64/FP32 = 1:3

Performance in function of the problem size

How does the performance scale with the number of events and particles?





Plots represent the time profiling for Tesla K40c.

Performance: OpenMP

The time spent for different numbers of OpenMP threads to process the 10 million events sample in a Xeon ES-2680 @2.5 GHz (24 physical cores, 48 logical cores):

	Time spent per number of threads (s)			
	#1	#12	#24	#48
$B^0 \to K\pi^+ J\psi$	3.65	0.369	0.218	0.161
$J\psi \to \mu^+\mu^-$	2.39	0.232	0.152	0.136
Dataset	1.69	0.33	0.33	0.271
$B^0 \to K\pi^+ J\psi$ w/ TGenPhaseSpace	4.68	-	-	-

For 10 million events with 9 particles in the final state:

- MCBooster takes 0.74 seconds using 24 OpenMP threads.
- TGenPhaseSpace takes 22 seconds.

Summary

- MCBooster is easy to integrate. See the poster Amplitude Analysis of four-body decays using a massively-parallel fitting framewor by C. Hasse et al.
- The project is released under the GNU General Public License version 3 and is hosted on GitHub: https://github.com/MultithreadCorner/MCBooster
- The package includes a suite of examples and an application to generate samples saved in ROOT TTrees.
- The next version will support the generation of samples corresponding to specific amplitude models.