Fermilab

Results of Review of Geant4 Electromagnetic Code

John Apostolakis/CERN, Andrea Dotti/SLAC, Krzysztof Genser/FNAL,
Soon Yung Jun/FNAL, Boyana Norris/Univ. of Oregon and ANL

presented by K. L. Genser (Review Coordinator)

HEP-ASCR Meeting
Fermilab, January 30th, 2015

Work supported in part through the US DOE
Advanced Scientific Computing Research (ASCR) program

Outline

- Scope, timeline
- Geant4 versions, tested applications and test tools
- Selected Specific Findings
- TAU highlights
- Selected general recommendations
- Summary

Scope, Timeline

- Review of performance aspects of a subset of Electromagnetic (EM) and closely related classes of Geant4 was performed to assess if the code was written in a computationally optimal way and to see if it could be improved
  - Keeping in mind
    - Correctness, performance, maintainability and adaptability
    - Multi-Threading aspects
- Concentrated on the most CPU costly classes and functions
- Review started in summer of 2013 an was mostly completed by June 2014 or earlier
  - Intermediate review results were presented at the 18th Geant4 Collaboration Meeting in September 2013, at the US HEP-ASCR Fermilab Meeting in February 2014 and at a joint meeting of the EM groups in March 2014
  - The final phase of the review was dedicated to writing of the report and doing analysis with TAU
  - Final results were presented at the 19th Geant4 Collaboration Meeting in September 2014
- Results and findings significantly affecting performance or physics were reported to the code maintainers on a regular basis
- Many of the class/function specific observations were mentioned earlier; only some of them are repeated here

Geant4 Versions and Tested Application

- Geant4 v9.6.r07 was the basis of the initial work; We worked with v10 in the final stages
- Settled on using SimplifiedCalo application as the executable which performance was analyzed to study effects of code transformations
  - It allowed to concentrate on the EM code and to minimize the importance other factors
    - If not doing full Geant4 profiling using PYTHIA Higgs (H → ZZ, Z → all decay channels) input events, most of the test were done with FTFP_BERT physics list ;50GeV e-; with no magnetic field
  - We have made sure that, while testing our changes, the final random number stayed the same after we modified the original code to ensure result invariance
- fast, TAU and IgProf profilers were used in analysis

TAU Results Highlights

- Type of object that invoked virtual functions of the base class
  - probably first analysis of this kind
    - including the next two items:
- FLOP inefficient functions
- Lightweight functions
  - with relatively few floating-point instructions per call but taking a significant fraction (> 0.5%) of the overall execution time

- See Boyana's talk for the details

Selected Specific Findings

G4PhysicsVector

- The G4PhysicsVector class is very computationally expensive (~10% of the total) and challenging at the same time
  – Code transformations we tried gave mixed results
    - See next two transparencies
  – Additional detailed studies are needed

G4PhysicsVector (in v9.6.r07)

- G4PhysicsVector ("container class" used heavily e.g. via G4PhysicsTable)
  - in order to collocate the data which is used together, replaced three main data members ("data", "bin", "secDerivative") of std::vector<G4double> type) with one std::vector<G4xyd>
    - G4xyd – a helper class with three G4double data members, operator<, and default () and 3 argument constructors(c'tors), (G4double x,G4double y,G4double d)
    - i.e. replaced three vectors with one vector of structs to localize access
  - modified derived classes accordingly
  - replaced hand-coded binary search in G4LPhysicsFreeVector::FindBinLocation with std::lower_bound
  - made other modifications following more current best coding practices (see last year's talk)

G4PhysicsVector (v9.6.r07 vs. v10)

- The overall effect of transforming G4PhysicsVector in v9.6.r07 was about 1.5% performance improvement as measured using standard profiling/benchmarking tools used to profile production/ reference Geant4 releases;
  - the G4PhysicsVector::Value function itself used to take about 3% of the total execution time so the decrease was a very significant fraction of that number
  - the timing improvement did not occur in that function itself but in other areas, mainly in CLHEP::MTwistEngine::flat likely due to cache effects
- No CPU effect or even a degradation in Geant4 v10 for the equivalent transformation was observed depending on the sample
  - in G4 v10, FindBinLocation was moved to the base class and inlined by the authors (an if was used to detect the type (using an enum) of the underlying classes)

G4PhysicsVector::FindBinLocation in v10

- We measured that the inlining of FindBinLocation in v10 caused about 1% performance degradation
  - There were event samples where it did slightly help
- Given that inlining of FindBinLocation may not be as beneficial as intended, we recommend abandoning the use of G4PhysicsVectorType and going back to using the virtual functions mechanism, delegating the type detection back to the compiler
  - We also noted a potential typo in one of the type enums in G4PhysicsLnVector where T_G4PhysicsLogVector is mixed with T_G4PhysicsLnVector (harmless in this specific case)
    - Using virtual functions avoids this type of problems

Selected General Suggestions

General comments

- Many of the findings/comments are not specific to the electromagnetic code

- Neither is the coding style which is more of a reflection of the time when Geant4 project was started and especially the state of C++ compilers (and STL in particular) which were available at that time when more “user code” had to be written than today

Adopt and follow Naming Conventions for Data Members and Global Variables

- Adopt a convention in the naming of data members and global variables
  - Lack of such a convention makes the code harder to read and maintain
    - Deep inheritance chains including user classes only exacerbate the situation

Rely on compilers and STL

- To eliminate unnecessary code maintenance and (likely) to improve performance
  - eliminate custom written copy constructors, assignment operators and destructors *when the compiler supplied ones are correct*
  - rely on the compiler type detection instead of hand coding the functionality (e.g. abandon using G4PhysicsVectorType like objects and rely on the virtual function mechanism instead)
  - use the ?: (ternary) operator
  - use more of the Standard Library algorithms when available (usually more efficient and well tested)
    - e.g. use std::lower_bound instead of hand written binary search (e.g. in the FindBinLocation functions

Optimize expressions to minimize number of calculations

- Minimize number of calculations by using known identities especially when transcendental functions are involved
  - optimization opportunities noted e.g. in G4UrbanMscModel

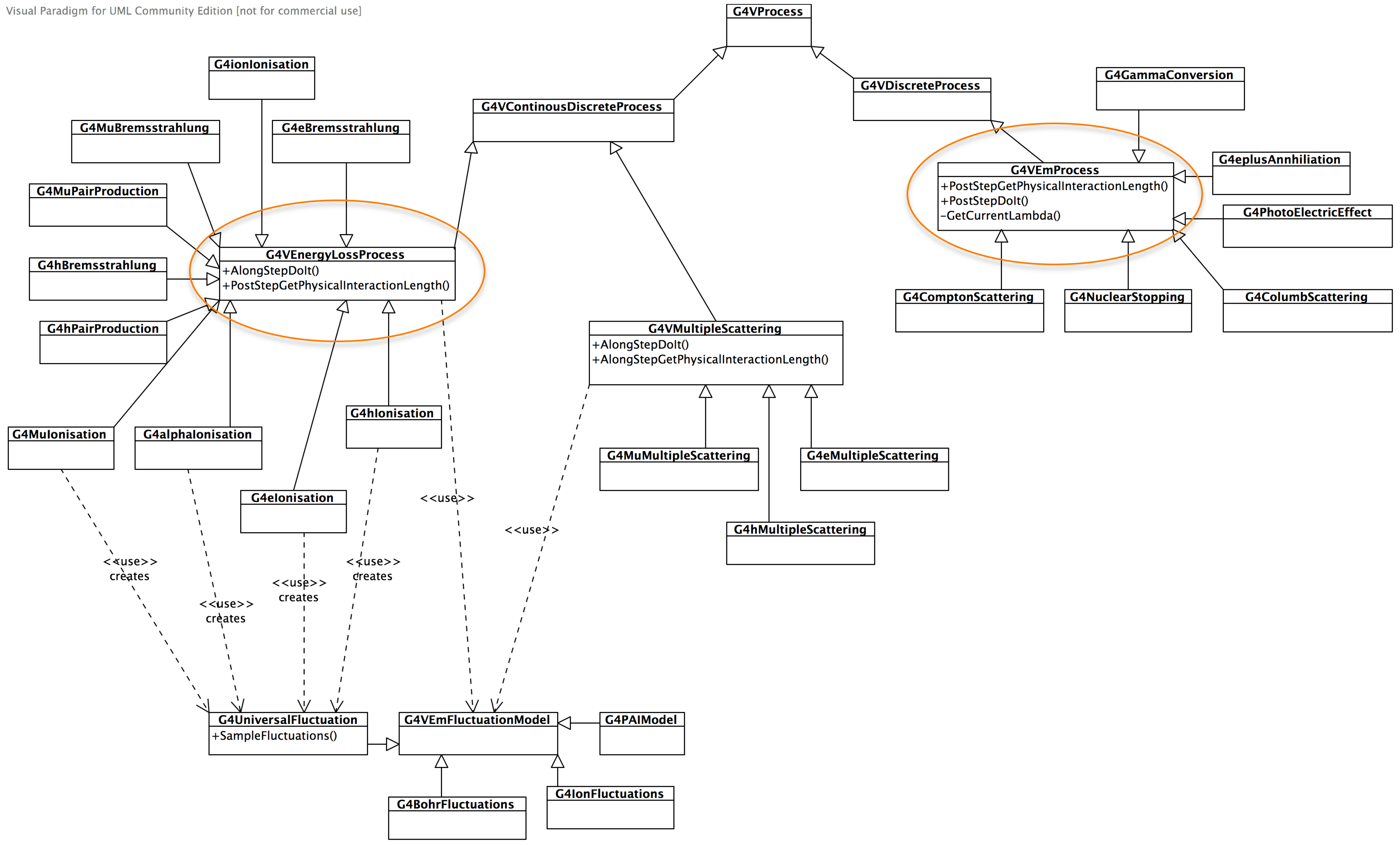
Inline frequently used functions

- Inline frequently used (short) functions
  - Like e.g. G4Poisson which resulted in ~2% gain
- Most of the CPU gains were indeed obtained by code inlining
  - Inlining needs to be done very judiciously as it does not always result in a performance gain
    - e.g. the case of G4PhysicsVector::FindBinLocation() where uninlining it in v10.0 resulted in about 1% gain in most samples
  - Use gcc -Winline option for diagnostics

Make code more intuitive

- Most electron processes inherit from G4VContinuousDiscreteProcess and yet behave as G4VDiscreteProcess
  - i.e. inherit from G4VEnergyLossProcess vs. G4VEmProcess
    - Could one move electron daughter classes from one to the other?
    - Two quite similar classes can they be consolidated?
  - A very non obvious code makes sure that “continuous” AlongStepDoIt functions are not called
- The class structure should be simplified or at least the code made more obvious
- Complicated side effects should be avoided or at least well documented

G4VProcess class diagram



Consider making G4PhysicsVector data members const

- Consider making G4PhysicsVector data members "read only" i.e. const (and initialize them in a constructor) to help with the MT code (if one can accommodate data retrieval from a file in an appropriate way)
  - rewrite deriving (from G4PhysicsVector) G4PhysicsOrderedFreeVector to use another data member as it modifies its data members after creation

Make constants const

- Remember to use static const when declaring class members when their values are never to change to utilize built in compiler capabilities

- Use global static const for physical constants etc... (e.g. from G4PhysicalConstants)
  - Especially important in the MT context

Revisit the use of Random Number Generators

- Use the array interface to enable future optimizations
    - remember that e.g. G4UniformRand() is a macro expanding to code returning one random number
- Cache the pointer to the random engine
- Pre-generate the numbers (locally or globally) when the needed number of them is known ahead of time
- Conduct a separate review of the use and generation of random numbers

Beware of impact of the changes

- Changing underlying data structures may have an impact comparable or larger than the fraction of the CPU taken by the functions using them
  - E.g. the change in G4Physics2Dvector from

    std::vector<std::vector<G4double>*> to

    std::vector<std::vector<G4double>   >

  Which in unit tests was promising (10%+improvement) but lead to a 5% overall degradation despite the fact that the function using the object was contributing only about 0.4% of the total CPU used by the executable

- All code changes should be verified by performing statistically significant profiling and benchmarking as some of the results may seem counter intuitive
  - likely due to the complicated interplay between the data structures/ layout and the algorithms operating on them as well as the ever evolving and becoming more sophisticated computing hardware

Verify new code

- Institute routine code inspections
  - It could be by a person chosen by the author of the (to be released) code, not necessarily by an external group
  - May help in reducing unintentional code features/behavior

Summary

- Compute intensive functions of EM Code and closely related functions were inspected visually, profiled with fast, IgProf and analyzed with TAU
- Performance gains based on the review recommendations are about 3%
  - Exact number is difficult to obtain as it fluctuated depending on the Geant4 release and changes in other areas of the toolkit
  - Inlining was the main source of the gains
- Found close similarities between G4VEmProcess and G4VEnergyLossProcess classes
  - Noted that some electron related G4VEnergyLossProcess classes should probably inherit from G4VDiscreteProcess (G4VEmProcess?)
- In addition to the performance and class structure related findings, a few small logic errors were found and brought to the attention of the authors
- The review stimulated development of new TAU capabilities
  - Future code analysis should benefit from it
  - Some of the characteristics were obtained for the first time and are unique to the review
- This talk only highlights some of the findings
  - The full report contains more details

Thanks

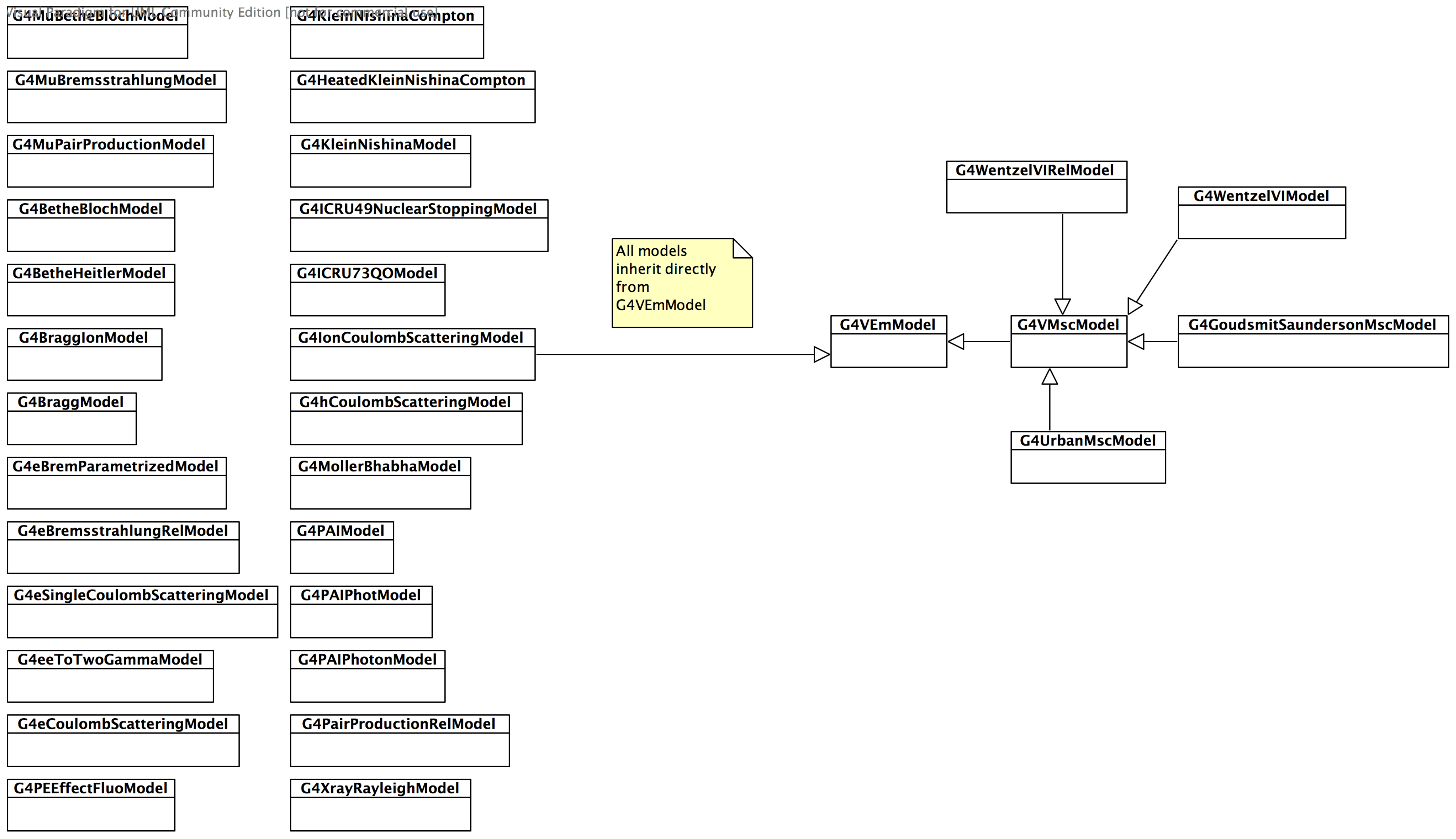
- We would like to thank all who assisted us in the review, helped improving the tools, read the report and provided comments, and especially to members of the EM Groups who interacted with us and Vladimir in particular

Backup Slides

Initial Plans

- Plan as formulated in Spring of 2013:
  - Review of performance aspects of a subset of ElectroMagnetic (EM) and closely related classes of Geant4 code with the initial goal to assess if the code is written in a computationally optimal way and to see if it could be improved, keeping in mind
    - correctness, performance, maintainability and adaptability
    - Multi-Threading aspects
    - potential issues related to parallelization and/or migration to GPUs
    - issues or potential improvements related to future migration to C++11
  - The review should initially concentrate on the most CPU costly classes and functions
  - After the initial phase it may be needed or useful to expand the scope of the review to other related areas or aspects of the code

G4VEmModel Class Diagram



Team

- Team members backgrounds and experiences cover various aspects of Geant4 and Computer Science
  - Geant4 itself
  - C++, source code analysis/transformation, performance tools, performance analysis, optimization
  - Profiling/Benchmarking
  - MultiThreading/GPUs/parallel code

- Mix of High Energy Physics and Computer Science backgrounds allows for interdisciplinary knowledge exchange and feedback also related to enhancement of code tuning and analysis tools

▪ Supported in part through US DOE joint High Energy Physics (HEP) and Advanced Scientific Computing Research (ASCR) collaboration

Initial List of Functions to be reviewed

- G4PhysicsVector, esp. (Compute)Value
  G4PhysicsLogVector, esp. FindBinLocation
- G4VProcess, esp. SubtractNumberOfInteractionLengthLeft
- G4VEmProcess, esp. PostStepGetPhysicalInteractionLength, GetCurrentLambda, PostStepDoIt
- G4VEnergyLossProcess esp. PostStepGetPhysicalInteractionLength, AlongStepDoIt, GetLambdaForScaledEnergy, AlongStepGetPhysicalInteractionLength
- G4VMultipleScattering esp. AlongStepDoIt, AlongStepGetPhysicalInteractionLength
- G4VEmModel, G4VMscModel, G4UrbanMscModel(95) esp. ComputeGeomPathLength, ComputeTruePathLengthLimit, SampleCosineTheta, SampleScattering

Test Clusters and Tools

- Main test cluster used for standard profiling/benchmarking
  - 5 nodes x 4x8 Core AMD Opteron Processor 6128 (CPU 2000 MHz)
  - L1 Cache Size 128KB, L2 Cache Size 512KB, L3 Cache Size 12288KB
  - 64GB memory on each node

- gcc v4.4..8 – v4.8.2

- fast (sampling) profiler
  - https://cdcvs.fnal.gov/redmine/projects/fast
  - https://oink.fnal.gov/perfanalysis/g4p/admin/task.html
    - Added new table to the standard profiling data to be used as an input to TAU and similar tools

- TAU (Tuning and Analysis Utilities) Performance System
  - With updates up to May 2014
  - Used on Intel Xeon X5650 2.67GHz 12-core dual processor nodes with 70GB of DRAM (per node)