IPCC ROOT

Princeton/Intel Parallel Computing Center

Showcase Presentation

Peter Elmer, Principal Investigator

Vassil Vassilev, Project Engineer

01.08.2017

Outline

- The ROOT project and its relevance for LHC and the field of high-energy physics
- IPCC-ROOT. Plan of work. Goals
- Code modernization:
  - Vectorization in ROOT's math libraries
  - Multi threaded file merging in ROOT's i/o libraries
  - Enabling automatic differentiation if ROOT's fitting libraries
- Future directions
- Other activities & Outreach

The LHC Data Analysis Toolkit ROOT

- Project started in 1995
- A few years later recognized by the biggest high-energy physics (HEP) labs: FNAL and CERN
- Approximately 10K active users
- Adopted in other fields such as finance, astronomy and biology

The LHC Data Analysis Toolkit ROOT

- Most HEP experiments' software depend on ROOT
- The HEP software which relies on ROOT is 100 M LOC
- ROOT multiple components such as io, math, gui, 2D and 3D graphics, neural nets, histograming and geometry
- Approximately 0.5-1.5 EB of data is stored in the ROOT data format

The plots presented at the Higgs boson discovery are produced by ROOT

ROOT Users

- **Physicists**
  - Programming skills vary dramatically
  - Quickly prototype a toy analysis, run it locally on small datasets, visualize results, potentially run the analysis on a farm, data center or a super computer

ROOT Users

- **Experiments**
  - Experts who ensure successful data taking from the machines
  - Sift the huge amounts of data (PB/s) and extract the ‘interesting’ physics
  - Store this ‘preprocessed’ data on the computing Grid ready to be processed and analyzed by physicists

ROOT Development

- 3.5 M LOC, mostly written in C++ and mostly under LGPL
- Over 200 contributors from all over the world with variety of backgrounds
  - Software developers from CERN and FNAL form the ROOT core team
- Over 300 releases, over 3.5K commits per year
- Recently ROOT moved to GitHub

ROOT Development. Contributors

The affiliations of contributors if the contributor disclosed it:

- Labs: ANL, BNL, DESY, FNAL, GSI, HZDR, INFN, JINR, KEK, LBL, NIKEF, RWTH, SLAC
- Universities: Bonn, Caltech, Karlsruhe, Chalmers, Cornell, John Hopkins, Princeton, Temple, Uppsala, Queen Mary, LMU, San Diego, Nebraska-Lincoln, etc
- Companies: the QT company, sutoiku, Yandex
- More

IPCC-ROOT

- ROOT is in the core of HEP experiments (including LHC's flagmen ALICE, ATLAS, CMS, LHCb). Even a small improvement in ROOT could have significant impact on the HEP community
- Princeton/Intel Parallel Computing Center to modernize ROOT funded via Intel's Parallel Computing Center (IPCC) program
- Started in 2017 in coordination with CERN OpenLab and the ROOT Team
- 1 full time engineer employed for 1 (+1) year, located at CERN, member of the ROOT team

Work plan 2017

Item	Deliverable	Success Criteria	Timeframe
Plan	Updated work plan for 2017	Approved work plan	Q1
ROOT Math	Integrate VecCore in ROOT. Help with ongoing math vectorization work.	Speed up the progress of vectorization of ROOT Math.	Q2
ROOT Math	Integrate the automatic differentiation prototype, clad, in ROOT.	Adoption in ROOT. Benchmark the performance of using it in fitting (minuit) or training neural networks (TMVA).	Q3
ROOT I/O	Thread-based file merging in ROOT based on a prototype in Geant by Witold Pokorski	Report and a prototype of the general concept.	Q4

Work plan 2017. Out-of-order Execution

Item	Deliverable	Success Criteria	Timeframe
Plan	Updated work plan for 2017	Approved work plan	Q1
ROOT Math	Integrate VecCore in ROOT. Help with ongoing math vectorization work.	Speed up the progress of vectorization of ROOT Math.	Q2
ROOT I/O	Thread-based file merging in ROOT based on a prototype in Geant by Witold Pokorski	Report and a prototype of the general concept.	Q3
ROOT Math	Integrate the automatic differentiation prototype, clad, in ROOT.	Adoption in ROOT. Benchmark the performance of using it in fitting (minuit) or training neural networks (TMVA).	Q4

Working Environment

Performance measurements are done on:

- Mac OS X, 2.5 GHz Intel Core i7, 16 GB
- CentOS 7.3 kernel 3.10.0-514.26.2.el7.x86_64, Intel Xeon CPU E5-2683 v3 @ 2.00GHz, 14 core (dual socket system => 14x2x2 = up to 56 logical), 64 GB DDR4, 2xSSDs 240GB (latest Haswell)
- CentOS 7.3 kernel 3.10.0-514.26.2.el7.x86_64, Intel Xeon Phi CPU 7210, 64 core (up to 256 logical) @ 1.30GHz, 16 GB MCDRAM, 96 GiB RAM DDR4, 4TB Disk + 240 GB SSD (latest KNL)

Code Modernization in ROOT. Vectorization

Integrate VecCore in ROOT. Help with ongoing math vectorization work.

Completed Q2 Deliverable (available in ROOT v6.10)

VecCore

- VecCore is a SIMD Vectorization Library which wraps Vc and UME::SIMD libraries. It is used in GeantV and was subsidized by the UNESP IPCC
- VecCore can be enabled in ROOT by passing -Dbuiltin_veccore=On in the build system

Code Modernization in ROOT. Vectorization

Integration of VecCore in ROOT enabled vectorization of other components in ROOT:

- ROOT's GenVector library
  - The role of IPCC-ROOT is to review pull requests, benchmark the code and further optimize bottlenecks
- ROOT's fitting libraries
  - The role of IPCC-ROOT is to give feedback and benchmark the relevant code in collaboration with the ROOT team

Uses

- LHCb experiment uses GenVector through the RICH mirror system
- Chris Jones (LHCb) presented some of their experience with vectorization and reported reduced time/event by 30%
- ROOT-IPCC took the work from a PR, reviewed it, tested and benchmarked it and added it to ROOT
- This made this experiment-specific contribution available to all experiments and users of ROOT

GenVector Performance: Synthetic Benchmarks

Performance on Haswell

Performance on KNL

The benchmark of GenVector describes very closely the use in LHCb.

GenVector Performance: Synthetic Benchmarks

Haswell: General Exploration. Summary

General Exploration General Exploration viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up

Elapsed Time: 52.389s

Clockticks:	147,372,000,000	
Instructions Retired:	123,728,000,000	
CPI Rate:	1.191	
MUX Reliability:	0.993	
Front-End Bound:	0.5%	of Pipeline Slots
Bad Speculation:	3.6%	of Pipeline Slots
Back-End Bound:	75.4%	of Pipeline Slots
Memory Bound:	20.8%	of Pipeline Slots
L1 Bound:	8.4%	of Clockticks
L2 Bound:	N/A with HT on	
L3 Bound:	N/A with HT on	
DRAM Bound:	N/A with HT on	
Memory Bandwidth:	N/A with HT on	
Memory Latency:	N/A with HT on	
Local DRAM:	56.3%	of Clockticks
Remote DRAM:	0.0%	of Clockticks
Remote Cache:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	54.6%	of Pipeline Slots
Divider:	33.9%	of Clockticks
Port Utilization:	54.9%	of Clockticks
Cycles of 0 Ports Utilized:	41.0%	of Clockticks
Cycles of 1 Port Utilized:	34.8%	of Clockticks

GCC6.2 performs slightly better in elapsed time and memory management

General Exploration General Exploration viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up

Elapsed Time: 54.169s

Clockticks:	146,880,000,000	
Instructions Retired:	137,188,000,000	
CPI Rate:	1.071	
MUX Reliability:	1.000	
Front-End Bound:	1.4%	of Pipeline Slots
Bad Speculation:	4.5%	of Pipeline Slots
Back-End Bound:	69.7%	of Pipeline Slots
Memory Bound:	38.6%	of Pipeline Slots
L1 Bound:	24.2%	of Clockticks
L2 Bound:	N/A with HT on	
L3 Bound:	N/A with HT on	
DRAM Bound:	N/A with HT on	
Memory Bandwidth:	N/A with HT on	
Memory Latency:	N/A with HT on	
Local DRAM:	63.3%	of Clockticks
Remote DRAM:	0.0%	of Clockticks
Remote Cache:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	31.0%	of Pipeline Slots
Divider:	34.3%	of Clockticks
Port Utilization:	28.9%	of Clockticks
Cycles of 0 Ports Utilized:	38.0%	of Clockticks
Cycles of 1 Port Utilized:	27.0%	of Clockticks

ICC17 performs slightly better in CPI rate and Core Bound

GenVector Performance: Synthetic Benchmarks

Haswell: General Exploration. Hotspots

reflectPlane<ROOT::Math::PositionVector3D<ROOT::Math::Cartesian3D<double>,	30.0%
ROOT::Math::Cartesian3D<double>::Scale	13.0%
ROOT::Math::Cartesian3D<double>::Scale	6.7%
_mm256_mul_pd	6.0%
reflectSpherical<ROOT::Math::PositionVector3D<ROOT::Math::Cartesian3D<doubl	39.3%
ROOT::Math::DisplacementVector3D<ROOT::Math::Cartesian3D<double>, ROOT:	5.8%
_mm256_mul_pd	5.0%
ROOT::Math::Cartesian3D<double>::Mag2	3.2%
ROOT::Math::Cartesian3D<double>::Mag2	3.0%
ROOT::Math::DisplacementVector3D<ROOT::Math::Cartesian3D<double>, ROOT:	3.0%
ROOT::Math::DisplacementVector3D<ROOT::Math::Cartesian3D<double>, ROOT:	5.2%
ROOT::Math::Cartesian3D<double>::SetXYZ	2.4%
ROOT::Math::Impl::Plane3D<double>::Distance	2.2%
_mm256_mul_pd	1.7%
main	99.8%
ROOT::Math::Impl::Plane3D<double>::Normal	1.5%
ROOT::Math::PositionVector3D<ROOT::Math::Cartesian3D<double>, ROOT::Math	1.5%

GCC6.2 has issues with _mm256_mul_pd

ROOT::Math::Cartesian3D<double>::Scale	12.6%
ROOT::Math::Cartesian3D<double>::Scale	11.4%
ROOT::Math::Cartesian3D<double>::Scale	9.4%
reflectSpherical<ROOT::Math::PositionVector3D<ROOT::Math::Cartesian3D<	43.7%
Vc_1::Detail::mul	5.8%
ROOT::Math::Impl::Plane3D<double>::Distance	5.6%
main	99.8%
ROOT::Math::DisplacementVector3D<ROOT::Math::Cartesian3D<double>, F	4.1%
Vc_1::Detail::mul	3.9%
ROOT::Math::Cartesian3D<double>::Mag2	3.6%
Vc_1::Detail::mul	3.6%
ROOT::Math::PositionVector3D<ROOT::Math::Cartesian3D<double>, ROOT	2.0%
__gnu_cxx::operator!=<Data<ROOT::Math::PositionVector3D<ROOT::Math::	1.7%
ROOT::Math::Cartesian3D<double>::SetXYZ	1.4%
ROOT::Math::Cartesian3D<double>::Mag2	1.2%
Vc_1::Detail::mul	1.2%
ROOT::Math::PositionVector3D<ROOT::Math::Cartesian3D<double>, ROOT	1.7%

ICC17 has issues with Vc::Detail::mul

Different compiler (version) different bottlenecks predominantly in the intentionally scalar part of the code.

GenVector Performance: Synthetic Benchmarks

Haswell: General Exploration. Roofline ICC17

GenVector Performance: Synthetic Benchmarks

Haswell: General Exploration. Roofline GCC62

GenVector Performance: Going a step further

- Adding #pragma omp parallel for directive to loop over the photons enables more efficient utilization of KNL

ICC17

Performance of Fitting Math Library

- Binned and unbinned likelihood fit functions are essential for minimization and fitting
- Work conducted by the ROOT team, in particular by Xavier Valls Pla as part of his PhD studies
- Feedback and profiling done by IPCC-ROOT

Future Work

- Enable -march=native in ROOT's C++ interpreter leveraging vector code
- Increase the micro benchmark coverage
- Track regressions with the micro benchmark infrastructure
- Continue profiling and improving the scalability of the code
- Continue to participate in the vectorization efforts of the ROOT team and others

Code Modernization in ROOT. Threading

Thread-based file merging in ROOT based on a prototype in Geant by Witold Pokorski

Completed Q3 Deliverable (available in ROOT v6.10)

Thread-based ROOT File Merging

- Scheduled for Q4. The ROOT team assessed its importance and decided to put it into the 6.10 release in June.
- The work was done in collaboration with the ROOT team

Code Modernization in ROOT. Threading

- The role of IPCC-ROOT was to outline the problem and the solution
- We participated in revamping the initial version of the code, finding a few bugs
- Guilherme Amadio took the responsibility to advance the code to its current state
- We participated in understanding the locks in the ROOT's reflection layer and implemented a few micro benchmarks helping us to understand the correlation between the auto flush size and number of threads

Thread-based ROOT File Merging

Enables multiple data writing threads into a single on-disk ROOT file.

```
//...
TBufferMerger merger("single_on_disk_file.root");
std::vector<std::thread> threads;
for (int i = 0; i < N; ++i) {
   threads.emplace_back([=, &merger]() {
      auto virt_file = merger.GetFile();
      auto mytree = new TTree("mytree", "mytree");
      Fill(mytree, i * nevents, nevents);
      virt_file->Write();
   });
}
for (auto &&t : threads) t.join();
//...
```

Thread-based ROOT File Merging. Micro benchmarks

We are still doing extra work

Reasonable scaling

Running TBufferMerger on KNL

Thread-based ROOT File Merging. Micro benchmarks

KNL: Concurrency. Hotspots

TVirtualMutex is heavily used to acquire locks.

Many thread transitions.

We should move the lock closer to the routine changing the state.

```
5477  void TClass::LoadClassInfo() const
5478  {
5479     R__LOCKGUARD(gInterpreterMutex);
5480
5481     // Return if another thread already loaded the info
5482     // while we were waiting for the lock
5483     if (!fCanLoadClassInfo)
5484        return;
5485
5486     bool autoParse = !gInterpreter->IsAutoParsingSuspended();
5487
5488     if (autoParse)
5489        gInterpreter->AutoParse(GetName());
5490
5491     if (!fClassInfo)
5492        gInterpreter->SetClassInfo(const_cast<TClass *>(this));
5493
```

Most frequent ‘client’ of TVirtualMutex is ROOT’s reflection layer. It acquires a lock.

Concurrency Hotspots by CPU Usage viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree

Grouping: Function / Call Stack

Function / Call Stack	CPU Time	Wait Time: Idle	Wait Time: Poor
__cxa_guard_acquire		4386.354s	1059.066s
_ZN10TLockGuardC4EP13TVirtualMutex	0.819s	19.016s	2137.563s
TClass::LoadClassInfo	0.819s	19.016s	2137.563s
_ZN10TLockGuardC4EP13TVirtualMutex	0.385s	1.180s	1113.727s
TStreamerInfo::Streamer ← TClass::Streamer ← TBufferFile::!	0.385s	1.180s	1113.727s
_ZN10TLockGuardC4EP13TVirtualMutex	0.420s	11.633s	975.499s
TClingClassInfo::GetBaseOffset ← TClass::GetBaseClassOffs	0.420s	11.633s	975.499s
std::condition_variable::wait	0.060s	104.905s	533.661s
std::condition_variable::wait<benchmark::Barrier::createBarrie	0.060s	90.860s	498.855s
std::condition_variable::wait<benchmark::Barrier::createBarrie		13.822s	21.095s
wait<ROOT::Experimental::TBufferMerger::WriteOutputFile()		0.224s	13.711s
_ZN10TLockGuardC4EP13TVirtualMutex	0.220s	0s	331.410s
_ZN10TLockGuardC4EP13TVirtualMutex	0.060s	13.227s	309.565s
_ZN10TLockGuardC4EP13TVirtualMutex	0.100s	16.922s	120.441s

Uses

- The CMS experiment has a mock-up of TBufferMerger just to be able to run improve the software in a multithreaded environment

- ROOT's new TDataFrame analysis infrastructure based on functional programming uses the TBufferMerger in its snapshot action

Summary and Conclusion

- New TBufferMerger class allows to write TTree in parallel
- Benchmarks performed on a dual-socket 18 core Xeon server at UNESP, more benchmarks to be run on Knights Landing
- Good performance compared with writing to multiple files
  No significant relative overhead up to ~30 threads
- Parallel snapshot action now available without changes in user code other than calling `ROOT::EnableImplicitMT()`
- However, some scaling issues remain for large numbers of worker threads, currently under investigation

Future Work

- Increase the micro benchmark coverage
- Track regressions with the micro benchmark infrastructure
- Reduce the amounts of locks and waits in the ROOT reflection layer
- Introduce a read write lock
- Continue profiling and improving the scalability of the third party code

Code Modernization in ROOT. Automatic Differentiation

Integrate the automatic differentiation prototype, clad, in ROOT.

Work in progress Q4 Deliverable (targeting ROOT v6.12)

Automatic Differentiation in a Nutshell. Clad

Automatic differentiation neither employs the slow symbolic nor inaccurate numerical differentiation. It uses the fact that every computer program can be divided into a set of elementary operations (-,+,*,/) and functions (sin, cos, log, etc). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed.

Clad is a C/C++ to C/C++ language transformer implementing the chain rule from differential calculus. For example:

```
constexpr double MyPow(double x) { return x*x; }
```

↓

```
constexpr double MyPow_darg0(double x) { return (1. * x + x * 1.); }
```

Integration Plan

- Enable the use of the library within ROOT, connecting it to the cling interpreter (also Clang/LLVM based), etc
- Update to the latest compiler versions, debug, etc
- Integrate AD into specific non-trivial examples in Minuit (used for numerical minimization in ROOT), TMVA (multivariate analysis) and machine learning in ROOT.
- Benchmark and profile

Advantages over Numerical Differentiation

Functions to differentiate.

```
#include <cmath>

double MyCos(double x) { return std::cos(x); }
double MySin(double x) { return std::sin(x); }
constexpr double MyPow(double x) { return x*x; }

// Simple finite differences numerical differentiator.
typedef double (*SigF)(double);
double derive(SigF f, double a, double h=0.01, double epsilon = 1e-7){
  double f1 = (f(a+h)-f(a))/h;
  double f2 = 0.;
  while (1) {
    h /= 2.;
    f2 = (f(a+h)-f(a))/h;
    double diff = std::abs(f2-f1);
    f1 = f2;
    if (diff < epsilon)
      break;
  }
  return f2;
}
```

Picking up a small step keeping roundoff errors under control depends on the differentiated function.

Advantages over Numerical Differentiation

```
#include <cmath>

double MyCos(double x) { return std::cos(x); }
double MySin(double x) { return std::sin(x); }
constexpr double MyPow(double x) { return x*x; }
```

Derivatives produced by clad.

```
// The derivatives are provided by clad but hardcoded here for
// simplicity, i.e. you can run this example without installing clad.
double MyCos_darg0(double x) { return -std::sin(x) * (1.); }
double MySin_darg0(double x) { return std::cos(x) * (1.); }
constexpr double MyPow_darg0(double x) { return (1. * x + x * 1.); }
```

Advantages over Numerical Differentiation

```c
// No clad, using the simple numerical differentiator
int main () {
  printf("MyCos' at 30 is %f\n", derive(MyCos, 30));
  // For every point we need to iterate :( This causes
  // not only slow execution but precision loss!
  printf("MyCos' at 31 is %f\n", derive(MyCos, 31));
  printf("MySin' at 30 is %f\n", derive(MySin, 30));

  // Even if MyPow is a compile-time foldable we still loop!
  printf("MyPow' at 2 is %f\n", derive(MyPow, 2));


  // From math we know that sinx' = cosx. Let's check.
  if (derive(MySin, 30) == MyCos(30))
    printf("No precision loss!\n");
  else
    printf("Precision loss!\n");

  // Output:
  //  MyCos' at 30 is 0.988032
  //  MyCos' at 31 is 0.404038
  //  MySin' at 30 is 0.154252
  //  MyPow' at 2 is 4.000000
  //  Precision loss!
  return 0;
}
```

Even this simple example yields precision loss.

```c
// Using clad, employing automatic differentiation techniques
int main () {
  printf("MyCos' at 30 is %f\n", MyCos_darg0(30));
  // For every point we just need to call a function
  // pointer!
  printf("MyCos' at 31 is %f\n", MyCos_darg0(31));
  printf("MySin' at 30 is %f\n", MySin_darg0(30));

  // The compile-time foldable MyPow folds away!
  printf("MyPow' at 2 is %f\n", MyPow_darg0(2));


  // From math we know that sinx' = cosx. Let's check.
  if (MySin_darg0(30) == MyCos(30))
    printf("No precision loss!\n");
  else
    printf("Precision loss!\n");

  // Output:
  //  MyCos' at 30 is 0.988032
  //  MyCos' at 31 is 0.404038
  //  MySin' at 30 is 0.154251
  //  MyPow' at 2 is 4.000000
  //  No precision loss!
  return 0;
}
```

clad has no problems, it returns the expected result

clang, gcc and icc generate 2-3x less assembly code

Application of clad in Machine Learning

$$y = \phi\left(\sum_i w_i x_i\right)$$

$$E = \frac{1}{2}(t-y)^2$$

$$\Delta w_i = \eta \frac{\partial E}{\partial w_i}$$

Training pattern is fed, forward generating corresponding output

Error at output, the error between observed and desired state. Computed from the output *y* and seen desired output *t*.

The error propagates back, through updates of the subtracted gradient ratio from the weights.

x_i, w_i, ϕ, η *are inputs, input weights, activation function and learning rate of the neuron*

Clad can provide efficient derivative computation reducing the CPU-intensive error propagation during training.

Extra work items

Completed (available in ROOT master)

Extra Things Delivered by IPCC-ROOT

- The regular nightly builds of ROOT and ICC17 were restored
- The ICC release builds now use default optimization level O2 (was O0)
- Optimization passes for runtime code (O2 in cling) were enabled
- Tools ensuring contribution quality such as clang-format, clang-tidy static analysis checks and clang-tidy modernization checks were enabled
- We reported and fixed a few build system issues when building in massively parallel mode with KNL

ROOT Build System Scalability

- ROOT now builds successfully on massively parallel machines
- IPCC-ROOT participated in discovering, reporting and fixing build system issues
- We could further improve the scalability by speeding up the I/O information generator and introducing .o level dependencies

ROOT builds ninja -j48 vs make -j48)

Future Directions

- Collaborate with the RooFit team and help with the redesign efforts especially vectorization and threading.
  - One of their major goals is to reduce Higgs combinations by orders of magnitude (from several hours to several minutes)
- Optimize ROOT's runtime and IO employing C++ Modules
  - Some of our synthetic benchmarks show 10 times faster execution and 2 times memory reduction. It can define away some of our locking restrictions and improve threading support in ROOT I/O.
- Integrate Matriplex into ROOT?

Other Activities & Outreach

Continuous efforts

Training — CoDaS-HEP school

A school on tools, techniques and methods for Computational and Data Science for High Energy Physics.

- First edition took place in Princeton University 10-13 July 2017
  - 40 participants
  - Topics included: performance tuning and optimization, vectorization, parallel programming (T. Mattson/Intel), and machine learning and big data tools.
- Second edition planned for summer 2018

Collaborating project — DIANA/HEP

An NSF-funded project focused on developing tools for the HEP analysis tools ecosystem (of which ROOT is a core element). DIANA/HEP has three broad goals: improving performance, increasing interoperability of HEP tools with the broader scientific software ecosystem and providing tools for collaborative analysis.

For the IPCC, the focus on performance is the relevant part. The IPCC will collaborate with DIANA (and the ROOT team) on I/O and probably (eventually) RooFit modernization.

Team: Princeton, U.Nebraska-Lincoln, U.Cincinnati, NYU

Website: http://diana-hep.org

Related projects — Parallel Kalman Filter Tracking

Charged particle tracking reconstruction is the key pattern recognition algorithm requiring modernization for parallel architectures and the challenges of the HL-LHC. This is an NSF-funded project which is aiming to modernize these algorithms for use by CMS and others at the HL-LHC.

For the IPCC project, it provides a key testbed and use cases for testing vectorization (e.g. Matriplex, VecGeom)

Team: Princeton, UCSD, Cornell

Website: http://trackreco.github.io

Thank you!

I'd like to thank Oksana Shadura, Guilherme Amadio, Raphael Isemann and the ROOT team for the help in various aspects from buying me coffee to contributing ideas & code;

Special thanks to Luca Atzori and CERN OpenLab for providing the cutting edge Intel infrastructure and technical support.

Backup Slides

Might look messier than expected.

C++ Modules Performance

Peak memory usage for ROOT's runtime

Code execution in ROOT's runtime

Further Reading About Clad

References:

[1] clad — Automatic Differentiation with Clang, http://llvm.org/devmtg/2013-11/slides/Vassilev-Poster.pdf
[2] clad Official GitHub Repository https://github.com/vgvassilev/clad
[3] clad demos https://github.com/vgvassilev/clad/tree/master/demos
[4] clad showcases https://github.com/vgvassilev/clad/tree/master/test
[5] More automatic differentiation tools http://www.autodiff.org/
[6] Automatic differentiation in Machine learning: a survey https://arxiv.org/pdf/1502.05767.pdf

TBufferMerger Plots

Compiler Compilation Flags

- g++ -pipe -m64 -Wshadow -Wall -W -Woverloaded-virtual -fsigned-char -fPIC -pthread -std=c++11 -DVECCORE_ENABLE_VC -DDNNCPU -O2 -g -DNDEBUG -rdynamic testGenVectorVc.cxx.o -o …
- icc -fPIC -wd1476 -wd1572 -m64 -wd279 -wd873 -wd2536 -wd597 -wd1098 -wd1292 -wd1478 -wd3373 -pthread -std=c++11 -DVECCORE_ENABLE_VC -DDNNCPU -O2 -g -DNDEBUG -rdynamic testGenVectorVc.cxx.o -o …

Uses

- CMS has a mock-up of TBufferMerger just to be able to run their software in a multithreaded environment

Uses

- ROOT's new TDataFrame analysis infrastructure based on functional programming uses it.

In process TTree Parallel Merge with new TBufferMerger class

G. Amadio and D. Piparo for the ROOT Team

Summary and Conclusion

- New TBufferMerger class allows to write TTree in parallel
- Benchmarks performed on a dual-socket 18 core Xeon server at UNESP, more benchmarks to be run on Knights Landing
- Good performance compared with writing to multiple files
  No significant relative overhead up to ~30 threads
- Parallel snapshot action now available without changes in user code other than calling `ROOT::EnableImplicitMT()`
- However, some scaling issues remain for large numbers of worker threads, currently under investigation

Pythia Event Generation: Speedup

GenVector Performance: Micro Benchmarks

- While the simple ray tracer scalability looks almost perfect (for SSE) there are still a few places which need improving
- We started benchmarking each function and found out some of them do not even compile if we pass the vector types.
- IPCC-ROOT is investing in building infrastructure which will continuously monitor performance

We are trying to understand this inconsistency.

Worldwide LCH Computing Grid

Data Workflow