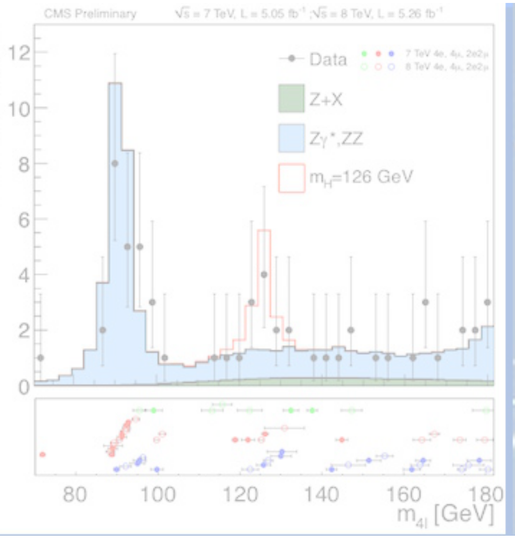


IPCC ROOT

Princeton/Intel Parallel Computing Center

Showcase Presentation

Vassil Vassilev, PhD

01.08.2017

IPCC-ROOT

- Princeton/Intel Parallel Computing Center to modernize ROOT funded via Intel's Intel Parallel Computing Center (IPCC) program.
- Started in 2017 in coordination with CERN OpenLab
- 1 full time engineer employed for 1 (+1) year, located at CERN

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

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

Completed

VecCore

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

Vectorization of GenVector Math Library

- Contribution originated from LHCb experiment based on Vc
- After many iterations of review and improvements it landed in ROOT and was released in the latest release 6.10

Benchmarks of GenVector

- A simple ray tracer using GenVector operations. Compares vectorized vs scalar mode while checking correctness

N traced photons	9.6K	96K	960K	9600K
SSE (gcc62)	1.8-2.5	2.02	1.73	1.7
AVX2 (gcc62)	2.5-3.3	2.8	1.82	1.77
SSE (icc17)	2.1	2.17	1.77	1.62
AVX2 (icc17)	2.3-3.6	2.8	1.85	1.75
KNL				
SSE (gcc62)	1.96	1.93	1.78	1.77
AVX2 (gcc62)	3.3	2.82	2.81	2.77
AVX 512	TODO	TODO	TODO	TODO

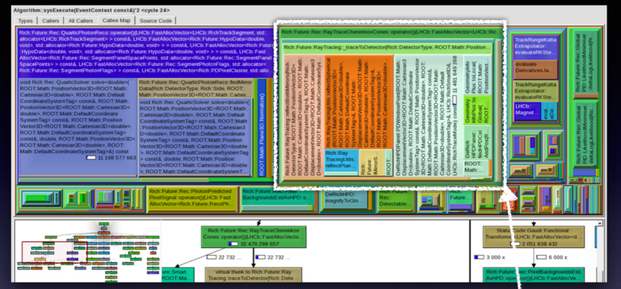
Benchmarks of GenVector. Micro Benchmarks

- 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.

Impact

- LHCb's uses GenVector through the RICH mirror system.
- Presentation by Chris Jones at a LHCb Software week

Introduction



- Ray tracing photons through RICH mirror system to detector plane major CPU usage for the RICH
- Simple geometrical calculation repeated many times. Should be easy to vectorise.
- Uses ROOT GenVector library.
- Have previously tried internally vectorising (vertical) the math libraries, using libraries like Eigen. Results not too impressive. Difficult to see how this approach can fully utilise SIMD capabilities.
- Look at using Vc library. Provides Vc::float_v, Vc::double_v types that behave like float, double, but are vectors of N values (N depends on SIMD level). Horizontal vectorisation.

Conclusions

- Vectorising the RICH ray tracing using GenVector+Vc works very well in the ideal case.
  - Get almost the perfect speed ups w.r.t. scalar.
  - Real world complications reduces this gain.
- Some temporary packages needed to bridge some missing functionality in LCG/ROOT.
  - Vc library - Should be in the next LCG release.
  - Improvements to GenVector - PR now submitted to ROOT.
- Longer term we probably should plan to migrate away from GenVector to something else (VecGeom) to fully utilise vectorisation in our basic math libraries.

Vectorization of Fitting Math Library

- Work conducted by the ROOT team, mostly by Xavier Valls Pla
- Binned and unbinned likelihood fit functions (some improvements landed in latest release 6.10)

Benchmarks of Fitting

- Performance of LogL unbinned likelihood function.

LogL	
SSE (gcc62)	1.62
AVX2 (gcc62)	3.37
SSE (icc17)	1.04
AVX2 (icc17)	2.13
KNL	
SSE (gcc62)	1.89
AVX2 (gcc62)	5.84
SSE(icc17)	1.23
AVX2 (icc17)	2.8
AVX 512	TODO

Benchmarks of Fitting. Micro Benchmarks

- If time permits we would like to add some micro benchmarks to ensure performance stability in a very fine grained manner

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

Completed

Thread-based ROOT File Merging

- Enable 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 myfile = merger.GetFile();
      auto mytree = new TTree("mytree", "mytree");
      Fill(mytree, i * nevents, nevents);
      myfile->Write();
   });
}
for (auto &&t : threads) t.join();
//...
```

Thread-based ROOT File Merging. Micro benchmarks

- Performance of writing a random data to TTree in parallel. Flush at 32Mb.

```
Run on (8 X 2500 MHz CPU s)
2017-07-15 14:43:50
------------------------------------------------------------------------------------------------------------
Benchmark                                                                  Time           CPU Iterations
------------------------------------------------------------------------------------------------------------
BM_TBufferFile_FillTreeWithRandomData_mean                              10005 us       7918 us           87
BM_TBufferFile_FillTreeWithRandomData_stddev                              251 us        191 us            0
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:8_mean           3005 us      10200 us          240
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:8_stddev          137 us        409 us            0
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:1_mean           9849 us       7792 us           77
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:1_stddev          245 us        193 us            0
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:2_mean           5877 us       7998 us          110
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:2_stddev          400 us        205 us            0
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:4_mean           3177 us       8635 us          212
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:4_stddev          104 us        254 us            0
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:8_mean           3561 us      12220 us          232
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:8_stddev          200 us        666 us            0
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:16_mean          3726 us      12949 us          208
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:16_stddev         145 us        432 us            0
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:32_mean          3697 us      14401 us          160
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:32_stddev         106 us        678 us            0
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:64_mean          3200 us      15703 us          192
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:64_stddev         142 us        507 us            0
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:128_mean         3260 us      17816 us          256
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:128_stddev        262 us       1109 us            0
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:256_mean         4517 us      26506 us          256
BM_TBufferFile_FillTreeWithRandomData/real_time/threads:256_stddev        634 us       3423 us            0
```

Thread-based ROOT File Merging

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

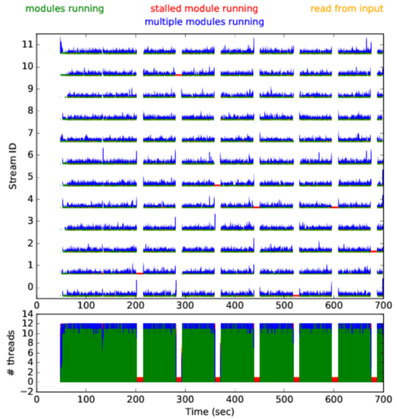
Impact

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

ROOT I/O limits CMS scaling

CMS production jobs are multithreaded

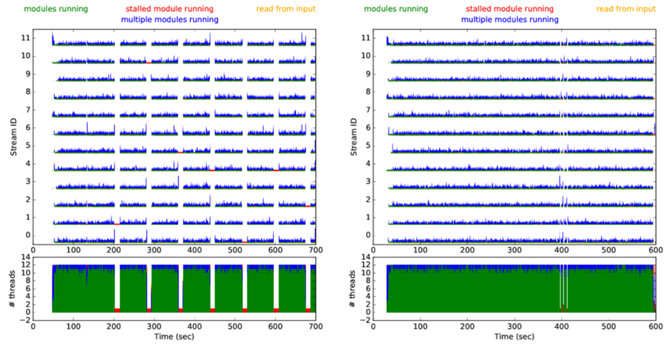
- Production jobs currently use 4 cores with 4 framework event streams
- Output is handled by "one" modules that can only be active on one thread at a time
- ROOT output is the dominant source of output stalls
  - We lose efficiency with more than 4 cores, preventing us going to 8 cores
- Compression is the principal bottleneck
  - Especially for AOD and MINIAOD data compress with LZMA



A biased comparison…

4 TMemFile intermediates, 134 MB flush size each

- Note scale change for right plot
- File size is ~15% larger
- RSS 280 MB larger than baseline, less than expected?
- TMemFile case only flushes twice (145MB output)



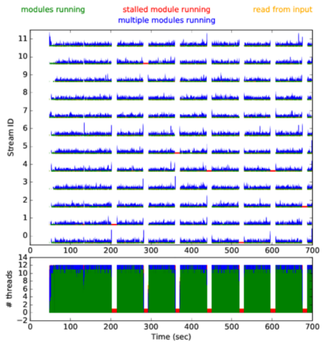
Baseline

4xTMemFile 134MB

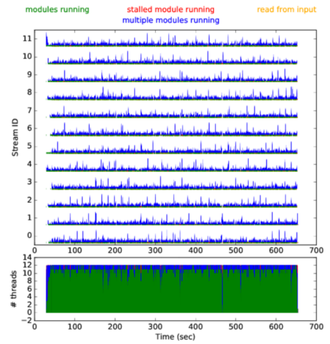
A fair comparison is…hard to find?

Naively

- Baseline is 15 MB autoflush size
- So if we have two TMemFile intermediates, try an 8 MB flush size
  - 8 MB blows up VSIZE and RSS!
  - Output file 40% larger
  - Does give an appreciable efficiency gain



Baseline



2xTMemFile 8MB

Impact

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

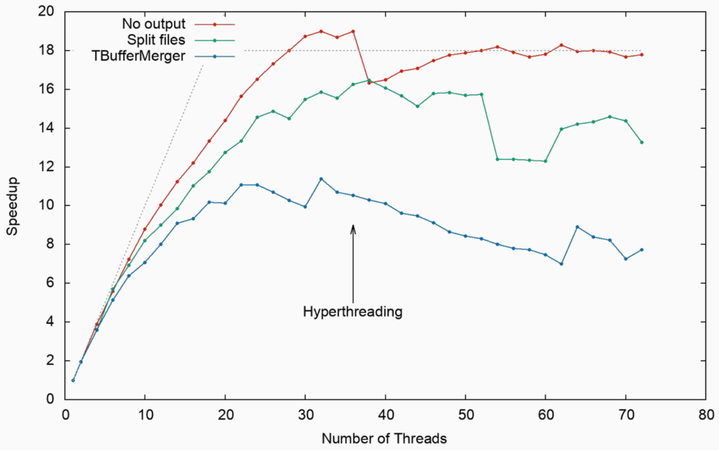
In process TTree Parallel Merge with new TBufferMerger class

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



Integrate the automatic differentiation prototype, clad, in ROOT.

Work in progress

clad: 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) and TMVA (multivariate analysis) in ROOT.
- Benchmark those applications

clad: In a Nutshell

clad 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 operation, derivatives of arbitrary order can be computed.

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.); }
```

clad: 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; }

typedef double (*SigF)(double);

// Simple finite differences numerical differentiator.
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;
}
```

clad: 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; }

// 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.); }
```

clad: Advantages over Numerical Differentiation

```
// 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 this was true.
  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;
}
```

Lines of assembly code

	-O0	-O3
gcc 6.1	223	141
clang 4	206	226
icc 17	279	283

```
// 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 this was true.
  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;
}
```

Lines of assembly code

	-O0	-O3
gcc 6.1	150	63
clang 4	154	65
icc 17	181	129

Further Reading

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/

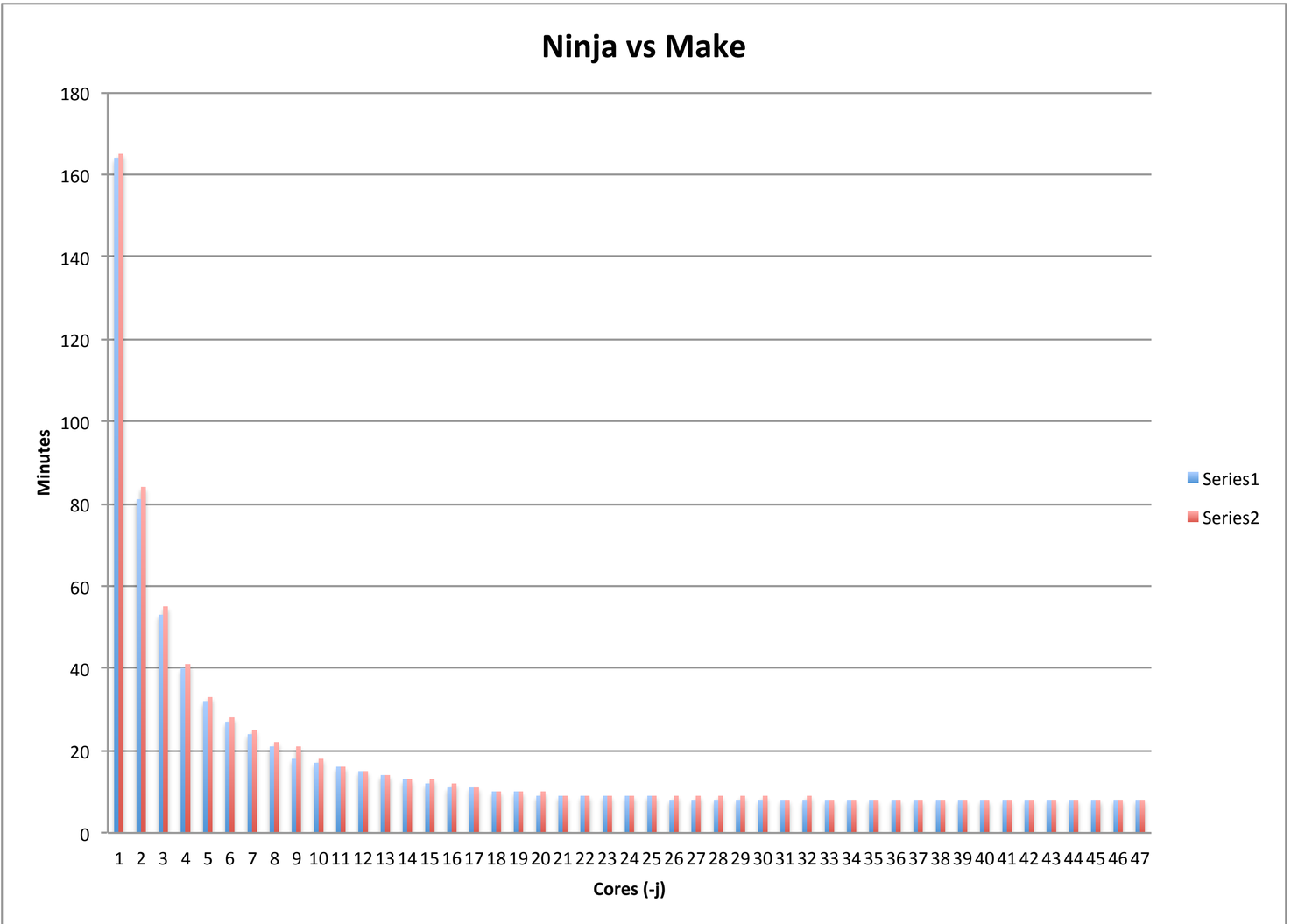
Extra work items

Completed

Extra Deliverables

- Restore the regular nightly builds of ROOT and ICC17
- Use O2 optimization level instead of O0 for ROOT release builds with ICC
- Enable optimization passes for runtime code (O2 in cling)
- Enable tools ensuring contribution quality such as clang-format, clang-tidy static analysis checks and clang-tidy modernization checks
- Report and fix a few build system issues when building in massively parallel mode with KNL

ROOT Build System



- ROOT's cmake build system when run with 48 parallel processes (ninja -j48 vs make -j48)
- We need to investigate what is the bottleneck
- Current theory is that ROOT I/O information generator is slow

Other potential interesting optimization targets

- Continuous integration of the benchmarks.
- Enabling -march=native in ROOT's C++ interpreter
- Modules. See Raphael's talk.

Thank you!

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