Optimizing ATLAS code with different profilers

S Kama[1], R Seuster[2], G A Stewart[3] and R A Vitillo[4]

[1] Southern Methodist University, Dallas, TX, USA
[2] TRIUMF, Vancouver BC V6T 2A3, Canada
[3] CERN and School of Physics and Astronomy, University of Glasgow, Glasgow G12 8QQ
[4] Lawrence Berkeley National Laboratory, USA

E-mail: `sami.kama@cern.ch`

Abstract. After the current maintenance period, the LHC will provide higher energy collisions with increased luminosity. In order to keep up with these higher rates, ATLAS software needs to speed up substantially. However, ATLAS code is composed of approximately 6M lines, written by many different programmers with different backgrounds, which makes code optimisation a challenge. To help with this effort different profiling tools and techniques are being used. These include well known tools, such as the Valgrind suite and Intel Amplifier; less common tools like Pin, PAPI, and GOoDA; as well as techniques such as library interposing. In this paper we will mainly focus on Pin tools and GOoDA. Pin is a dynamic binary instrumentation tool which can obtain statistics such as call counts, instruction counts and interrogate functions' arguments. It has been used to obtain CLHEP Matrix profiles, operations and vector sizes for linear algebra calculations which has provided the insight necessary to achieve significant performance improvements. Complimenting this, GOoDA, an in-house performance tool built in collaboration with Google, which is based on hardware performance monitoring unit events, is used to identify hot-spots in the code for different types of hardware limitations, such as CPU resources, caches, or memory bandwidth. GOoDA has been used in improvement of the performance of new magnetic field code and identification of potential vectorization targets in several places, such as Runge-Kutta propagation code.

1. Introduction

The Large Hadron Collider (LHC) [1] is a proton collider built about 100m underground near Geneva, Switzerland. It has a 27 km circumference and it is designed to collide protons every 25 ns at a center-of-mass energy of 14 TeV with a luminosity of 10^{34} cm^{-2} s^{-1}. It started operation in 2010 and gradually increased the collision energy and luminosity. In 2011 LHC delivered a peak luminosity of 3.42×10^{33} cm^{-2} s^{-1} in 31 weeks of proton-proton collision runs at $\sqrt{s}$ = 7 TeV with 50 ns bunch crossing interval. In 2012, the center-of-mass energy was increased to 8 TeV and the peak instantaneous luminosity exceeded 7.7×10^{33} cm^{-2} s^{-1}. From March 2013, it has been shutdown for two years for maintenance and upgrades. It will operate at a higher beam energy and higher luminosity after the shutdown.

There are 4 detectors located at the LHC. ATLAS [2] is one of the two large general purpose detectors. It is composed of different co-centric cylindrical detectors of about 100 M readout channels in total. The ATLAS detector has a three-level trigger system to do event selection and reduce the enormous amounts of data produced to manageable sizes. Level 1 of the trigger is based on hardware and located on the detector. It has a design input rate of 40 MHz and an output rate of 75 kHz. Level 2 and Level 3 are based on software running on a PC farm of about 16k cores. They reduce final event rate to 600 Hz at ~1.6 MB per event. Selected events are stored and processed offline in more detail. Both online and offline selection is done using the same software framework called Athena, with different configurations. Up to the shutdown in 2013 ATLAS had stored and processed ~22 PB of raw data.

2. ATLAS Software

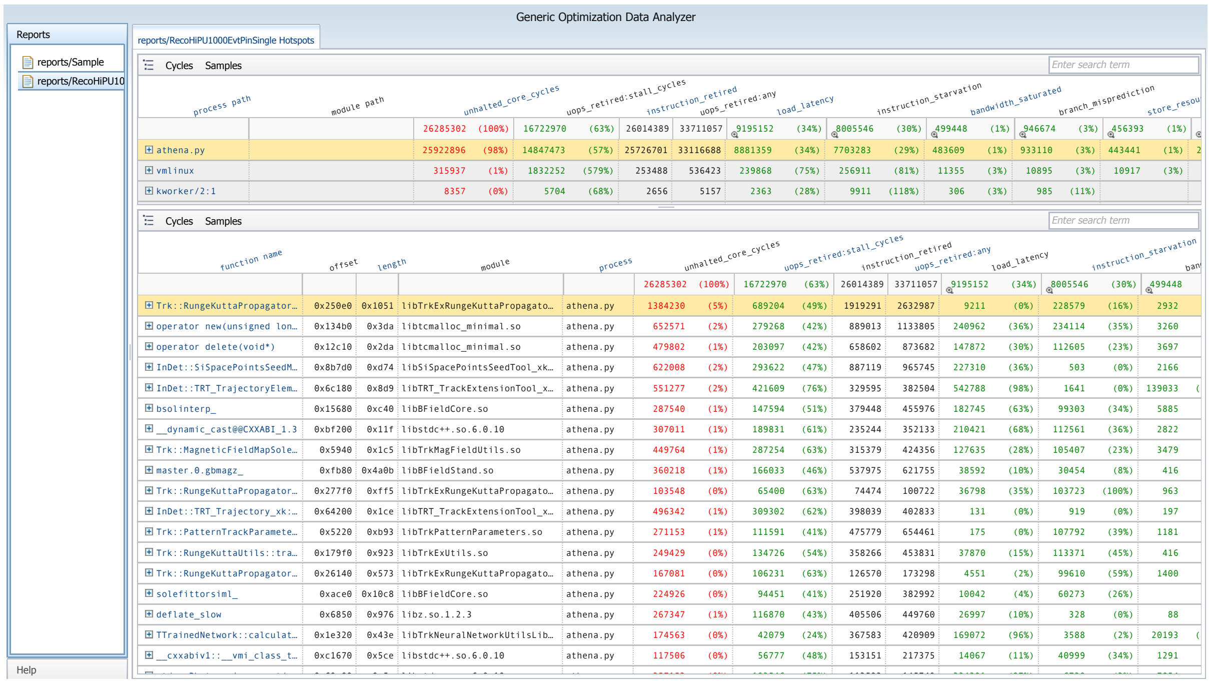
The ATLAS software framework is based on the Gaudi framework [3] and comprises more than 6 million lines of C++ and Python, with a small amount of FORTRAN code. It is spread over about 2000 packages, producing more than 4000 libraries of various sizes. It has been evolving for more than 10 years and was written by people with varied programming backgrounds. Some parts of the code were written by expert level programmers, while some were written by people with minimal programming knowledge. Throughout its development, detailed knowledge of packages has been frequently lost due to authors and maintainers changing topics, institutes or leaving the field. Athena configuration is done in Python and reconstruction of a 2012 high pile-up sample($\langle\mu\rangle \sim 35$) with 64-bit application consumes about ~4 GB of virtual memory and has approximately 2.7 GB resident set size(RSS).

With the increase in LHC energy, collision rate, event complexity and trigger output, ATLAS software needs to speed up considerably. Due to sheer size of the codebase and the domain expertise required it is not practical to go through the codebase to fix even the simplest perfomance issues. So instead, efforts should be focused on so called hot-spots, where a small section of source code contributes significantly to overall application run time. In order to monitor the performance of Athena, ATLAS uses various tools. These range from commonly available tools such as Valgrind suite, Google perf tools, oprofile, Intel VTUNE, igprof, PAPI and Pin [4] to in house developed tools PerfMon and Generic Optimization Data Analyzer (GOoDA) [5], which is developed in collaboration with Google. However, the enormous size of Athena creates significant challanges for most of these tools. In this report we will be describing our experience of improving the performance of Athena using GOoDA and Pin.

3. GOoDA

The Generic Optimization Data Analyzer is an open source project developed by a collaboration between ATLAS and Google. It uses Linux `perf` tool to configure and collect detailed performance monitoring unit(PMU) information from hardware monitoring units built in CPUs. Then it analyses the collected monitoring information, grouping it into a few issue classes. These issue classes provide a more general understanding of the code's performance than CPU specific PMU values. Results of the analysis are written in report spreadsheets which can we viewed through a web browser either locally or remotely. An example of such a report is shown in figure 1.

GOoDA reports contain hot-spots for each type of performance issue down to instruction level. It can also display source line if sources are available. Figure 1 shows a profile of Athena offline reconstruction job with high number of proton collisions (pile-up) per input event. Investigation of the profile shows several groups of code with rather specific issues. The most frequently encountered code domain is the tracking code. Being composed of a high number of arithmetic calculations, due to vector and matrix operations, this code frequently has instruction related issues, such as instruction starvation. It is possible to overcome such issues by using *single instruction multiple data* (SIMD) operations. Due to the intrinsic geometric nature of the tracking problem, representing tracking operations in terms of primitives of a SIMD vector math library is the most feasible approach, both from maintainability and development time points of view. However, such vector math libraries are not equally efficient for all types of operations or for all sizes of vectors and matrices. How the decision on which vector math library to use was taken is explained in the next section.


Generic Optimization Data Analyzer
Reports
reports/Sample
reports/RecoHiPU10
reports/RecoHiPU1000EvtPinSingle Hotspots
Cycles
Samples
Enter search term
Help

process path	module path	unhalted_core_cycles	uops_retired:stall_cycles	instruction_retired	uops_retired:any	load_latency	instruction_starvation	bandwidth_saturated	branch_misprediction	store_resou	
		26285302 (100%)	16722970 (63%)	26014389	33711057	9195152 (34%)	8005546 (30%)	499448 (1%)	946674 (3%)	456393 (1%)	
athena.py		25922896 (98%)	14847473 (57%)	25726701	33116688	8881359 (34%)	7703283 (29%)	483609 (1%)	933110 (3%)	443441 (1%)	2
vmlinux		315937 (1%)	1832252 (579%)	253488	536423	239868 (75%)	256911 (81%)	11355 (3%)	10895 (3%)	10917 (3%)	
kworker/2:1		8357 (0%)	5704 (68%)	2656	5157	2363 (28%)	9911 (118%)	306 (3%)	985 (11%)		

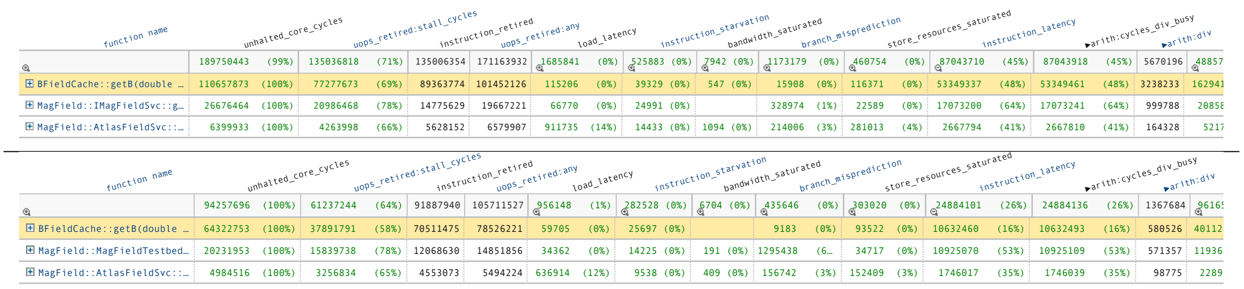
function name	offset	length	module	process	unhalted_core_cycles	uops_retired:stall_cycles	instruction_retired	uops_retired:any	load_latency	instruction_starvation	ban
					26285302 (100%)	16722970 (63%)	26014389	33711057	9195152 (34%)	8005546 (30%)	499448
Trk::RungeKuttaPropagator…	0x250e0	0x1051	libTrkExRungeKuttaPropagato…	athena.py	1384230 (5%)	689204 (49%)	1919291	2632987	9211 (0%)	228579 (16%)	2932
operator new(unsigned lon…	0x134b0	0x3da	libtcmalloc_minimal.so	athena.py	652571 (2%)	279268 (42%)	889013	1133805	240962 (36%)	234114 (35%)	3260
operator delete(void*)	0x12c10	0x2da	libtcmalloc_minimal.so	athena.py	479802 (1%)	203097 (42%)	658602	873682	147872 (30%)	112605 (23%)	3697
InDet::SiSpacePointsSeedM…	0x8b7d0	0xd74	libSiSpacePointsSeedTool_xk…	athena.py	622008 (2%)	293622 (47%)	887119	965745	227310 (36%)	503 (0%)	2166
InDet::TRT_TrajectoryElem…	0x6c180	0x8d9	libTRT_TrackExtensionTool_x…	athena.py	551277 (2%)	421609 (76%)	329595	382504	542788 (98%)	1641 (0%)	139033 (
bsolinterp_	0x15680	0xc40	libBFieldCore.so	athena.py	287540 (1%)	147594 (51%)	379448	455976	182745 (63%)	99303 (34%)	5885
__dynamic_cast@@CXXABI_1.3	0xbf200	0x11f	libstdc++.so.6.0.10	athena.py	307011 (1%)	189831 (61%)	235244	352133	210421 (68%)	112561 (36%)	2822
Trk::MagneticFieldMapSole…	0x5940	0x1c5	libTrkMagFieldUtils.so	athena.py	449764 (1%)	287254 (63%)	315379	424356	127635 (28%)	105407 (23%)	3479
master.0.gbmagz_	0xfb80	0x4a0b	libBFieldStand.so	athena.py	360218 (1%)	166033 (46%)	537975	621755	38592 (10%)	30454 (8%)	416
Trk::RungeKuttaPropagator…	0x277f0	0xff5	libTrkExRungeKuttaPropagato…	athena.py	103548 (0%)	65400 (63%)	74474	100722	36798 (35%)	103723 (100%)	963
InDet::TRT_Trajectory_xk:…	0x64200	0x1ce	libTRT_TrackExtensionTool_x…	athena.py	496342 (1%)	309302 (62%)	398039	402833	131 (0%)	919 (0%)	197
Trk::PatternTrackParamete…	0x5220	0xb93	libTrkPatternParameters.so	athena.py	271153 (1%)	111591 (41%)	475779	654461	175 (0%)	107792 (39%)	1181
Trk::RungeKuttaUtils::tra…	0x179f0	0x923	libTrkExUtils.so	athena.py	249429 (0%)	134726 (54%)	358266	453831	37870 (15%)	113371 (45%)	416
Trk::RungeKuttaPropagator…	0x26140	0x573	libTrkExRungeKuttaPropagato…	athena.py	167081 (0%)	106231 (63%)	126570	173298	4551 (2%)	99610 (59%)	1400
solefittorsiml_	0xace0	0x10c8	libBFieldCore.so	athena.py	224926 (0%)	94451 (41%)	251920	382992	10042 (4%)	60273 (26%)	
deflate_slow	0x6850	0x976	libz.so.1.2.3	athena.py	267347 (1%)	116870 (43%)	405506	449760	26997 (10%)	328 (0%)	88
TTrainedNetwork::calculat…	0x1e320	0x43e	libTrkNeuralNetworkUtilsLib…	athena.py	174563 (0%)	42079 (24%)	367583	420909	169072 (96%)	3588 (2%)	20193 (
__cxxabiv1::__vmi_class_t…	0xc1670	0x5ce	libstdc++.so.6.0.10	athena.py	117506 (0%)	56777 (48%)	153151	217375	14067 (11%)	40999 (34%)	1291

Figure 1. Main page of GOoDA profile. The left frame displays available reports. The top right frame displays processes in the report and their respective PMU event counts. The bottom right frame shows list of hot-spot functions in the selected process and their respective event counts.

The second group is the frequent use of new/delete operators, which is due to the design of the event data model. Frequent memory allocation and deallocation was already identified previously and some actions, such as replacing standard malloc with thread caching malloc (tcmalloc) from Google, were used to mitigate the effect of this. However, it still remains a significant problem and the profile helps to quantify it. In order to further reduce these frequent allocation/deallocation effects, there have been some changes in the event data model. There are also some studies, such as using memory arenas and different memory allocators, ongoing to reduce the impact on the performance.

The third group is the magnetic field code. This was written in FORTRAN and most of the time was actually spent on interfacing the old FORTRAN code with the C++ framework. So, rather than optimizing FORTRAN, the code was re-written in C++ from scratch. The new code was already 2x faster than the FORTRAN implementation due to a better design and native interfacing. After profiling a simple test job, which is querying the magnetic field at random points in the detector to stress it, further hot-spots were identified (see figure 2). The most notable point in this table is that about 70% of stall cycles are due to instruction latency coming from division operations. Following up the report points to the source lines 101-108 in figure 3. However, even though replacing division operators with inverse multiplications is straight forward and common performance practice, it doesn't maximize the performance. Further profiles have shown that there is instruction starvation around the lines shown in figure 3. After a detailed investigation of the source code, it can be seen that the complex calculations in lines 84-95 can be written as a dot product of two vectors with properly arranged coefficients. Profiling the version with the vector products and inverse multiplications is shown in the bottom table in figure 2. It is clearly visible here, that total unhalted core cycles count is reduced by 42%

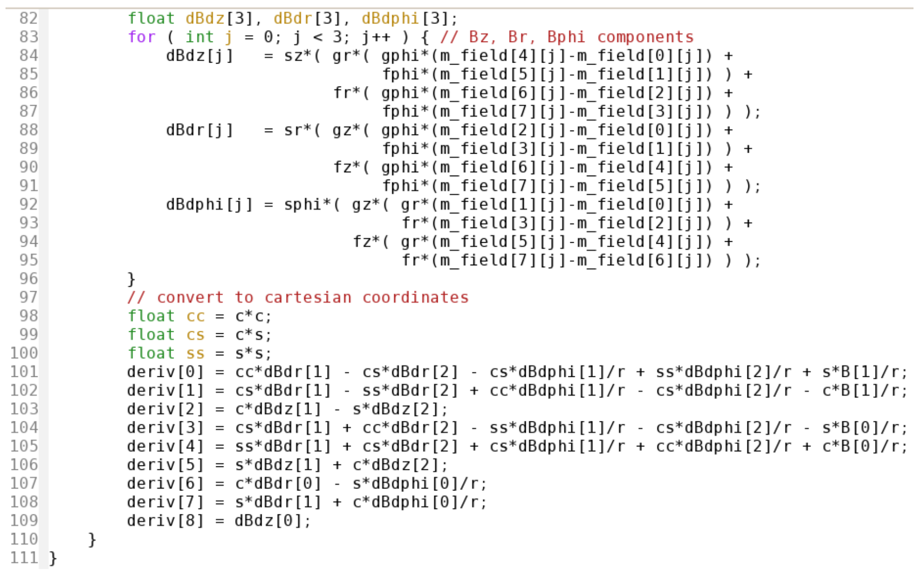
with respect to initial profile. The final version is about 40% faster than initial code and overall effect of the new code with respect to the FORTRAN implementation is measured between 5% to 20% improvement in the runtime of simulation jobs.



function name	unhalted_core_cycles	uops_retired:stall_cycles	instruction_retired	uops_retired:any	load_latency	instruction_starvation	bandwidth_saturated	branch_misprediction	store_resources_saturated	instruction_latency	▸arith:cycles_div_busy	▸arith:div
	189750443 (99%)	135036818 (71%)	135006354	171163932	1685841 (0%)	525883 (0%)	7942 (0%)	1173179 (0%)	460754 (0%)	87043710 (45%)	87043918 (45%)	5670196	48857
⊞ BFieldCache::getB(double …	110657873 (100%)	77277673 (69%)	89363774	101452126	115206 (0%)	39329 (0%)	547 (0%)	15908 (0%)	116371 (0%)	53349337 (48%)	53349461 (48%)	3238233	162941
⊞ MagField::IMagFieldSvc::g…	26676464 (100%)	20986468 (78%)	14775629	19667221	66770 (0%)	24991 (0%)		328974 (1%)	22589 (0%)	17073200 (64%)	17073241 (64%)	999788	20858
⊞ MagField::AtlasFieldSvc::…	6399933 (100%)	4263998 (66%)	5628152	6579907	911735 (14%)	14433 (0%)	1094 (0%)	214006 (3%)	281013 (4%)	2667794 (41%)	2667810 (41%)	164328	5217

function name	unhalted_core_cycles	uops_retired:stall_cycles	instruction_retired	uops_retired:any	load_latency	instruction_starvation	bandwidth_saturated	branch_misprediction	store_resources_saturated	instruction_latency	▸arith:cycles_div_busy	▸arith:div
	94257696 (100%)	61237244 (64%)	91887940	105711527	956148 (1%)	282528 (0%)	6704 (0%)	435646 (0%)	303020 (0%)	24884101 (26%)	24884136 (26%)	1367684	96165
⊞ BFieldCache::getB(double …	64322753 (100%)	37891791 (58%)	70511475	78526221	59705 (0%)	25697 (0%)		9183 (0%)	93522 (0%)	10632460 (16%)	10632493 (16%)	580526	40112
⊞ MagField::MagFieldTestbed…	20231953 (100%)	15839738 (78%)	12068630	14851856	34362 (0%)	14225 (0%)	191 (0%)	1295438 (6…	34717 (0%)	10925070 (53%)	10925109 (53%)	571357	11936
⊞ MagField::AtlasFieldSvc::…	4984516 (100%)	3256834 (65%)	4553073	5494224	636914 (12%)	9538 (0%)	409 (0%)	156742 (3%)	152409 (3%)	1746017 (35%)	1746039 (35%)	98775	2289

Figure 2. Profile report of new magnetic field test job. The top profile is before optimization. The first hot-spot shows high percentage of stall cycles mostly originating from instruction latency coming from division operations. The bottom profile is after optimizing divisions and multiplications.



```
        float dBdz[3], dBdr[3], dBdphi[3];
        for ( int j = 0; j < 3; j++ ) { // Bz, Br, Bphi components
            dBdz[j]   = sz*( gr*( gphi*(m_field[4][j]-m_field[0][j]) +
                                  fphi*(m_field[5][j]-m_field[1][j]) ) +
                             fr*( gphi*(m_field[6][j]-m_field[2][j]) +
                                  fphi*(m_field[7][j]-m_field[3][j]) ) );
            dBdr[j]   = sr*( gz*( gphi*(m_field[2][j]-m_field[0][j]) +
                                  fphi*(m_field[3][j]-m_field[1][j]) ) +
                             fz*( gphi*(m_field[6][j]-m_field[4][j]) +
                                  fphi*(m_field[7][j]-m_field[5][j]) ) );
            dBdphi[j] = sphi*( gz*( gr*(m_field[1][j]-m_field[0][j]) +
                                    fr*(m_field[3][j]-m_field[2][j]) ) +
                               fz*( gr*(m_field[5][j]-m_field[4][j]) +
                                    fr*(m_field[7][j]-m_field[6][j]) ) );
        }
        // convert to cartesian coordinates
        float cc = c*c;
        float cs = c*s;
        float ss = s*s;
        deriv[0] = cc*dBdr[1] - cs*dBdr[2] - cs*dBdphi[1]/r + ss*dBdphi[2]/r + s*B[1]/r;
        deriv[1] = cs*dBdr[1] - ss*dBdr[2] + cc*dBdphi[1]/r - cs*dBdphi[2]/r - c*B[1]/r;
        deriv[2] = c*dBdz[1] - s*dBdz[2];
        deriv[3] = cs*dBdr[1] + cc*dBdr[2] - ss*dBdphi[1]/r - cs*dBdphi[2]/r - s*B[0]/r;
        deriv[4] = ss*dBdr[1] + cs*dBdr[2] + cs*dBdphi[1]/r + cc*dBdphi[2]/r + c*B[0]/r;
        deriv[5] = s*dBdz[1] + c*dBdz[2];
        deriv[6] = c*dBdr[0] - s*dBdphi[0]/r;
        deriv[7] = s*dBdr[1] + c*dBdphi[0]/r;
        deriv[8] = dBdz[0];
    }
}
```

Figure 3. Source code of hot-spots in the magnetic field code. Division operations are the cause of the instruction latency and complex calculations are the cause of instruction starvation. These complex calculations can be re-written as a dot product of two vectors.

4. Pin Tools

Pin is a dynamic binary instrumentation framework from Intel. It instruments binaries at run-time, eliminating need to modify or recompile the code. It can instrument the binary from instruction level to function level and supports dynamically generated code. Pin can access function parameters and register contents and it can work with threaded programs. If available, it provides limited access to symbol and debug information in the original binary. Instrumentation is done on an in-memory copy of the binary. Pin inspects the instructions in original binary and inserts calls to analysis functions. Due to its powerful features it has been used in computer architecture, security, emulation and parallel program analysis tools, such as Intel's Parallel Inspector, Parallel Amplifier, Trace Analyzer and Collector as well as CMP$im and many others. It has detailed documentation and an active user support community called PinHeads.

As mentioned in previous section, tracking code is largely composed of vector and martix operations. In Athena, CLHEP library [6] is used for most of the vector and matrix representation and their operations. However, the CLHEP library is not performance optimized and does not vectorize well. There are other SIMD vector and matrix libraries available, each having different performance for different primitive sizes and operations. In order to quantify most common vector dimensions and operation we instrumented CLHEP vector classes in Athena with Pin.

Table 1. Top 10 most frequently called CLHEP functions in Athena for 20 events with total instructions and average instructions per call, sorted by total instruction count.

Calls	Instr	<instr>/call	Function
1778523	6392431813	3594.24	operator*(HepMatrix const&, HepSymMatrix const&)
671676353	5988139520	8.92	Hep2Vector::operator()(int) const
232093102	5956556656	25.66	Transform3D::operator()(int, int) const
285282108	3709057782	13.00	Hep3Vector::operator()(int)
15815930	3179001930	201.00	HepRotation::rotateAxes(Hep3Vector const&, Hep3Vector const&, Hep3Vector const&)
20529818	2422518524	118.00	Transform3D::inverse() const
31612743	2212258670	69.98	HepSymMatrix::HepSymMatrix(HepSymMatrix const&)
28914115	1929106393	66.72	HepVector::HepVector(int, int)
51974716	1819115060	35.00	operator*(Transform3D const&, Point3D<double>const&)
27652274	1506352669	54.47	HepVector::HepVector(HepVector const&)

Table 1 shows the most frequently called CLHEP functions, together with their instruction costs and averge number of instructions per event, sorted by their total instruction counts. The multiplication operator for multiplication of a matrix with a symmetric matrix is the most expensive operation. These matrices usually have dimensions 5×5, 5×3, 3×3 and 3×5. Accessor operators for vectors and 3D transformation matrices also contribute significantly to instruction counts. The number of instructions can be significantly reduced by inlining.

Using information gained from Pin instrumentation, we devised a benchmark for various SIMD vector math libraries. The benchmark composes several operations such as $A_{5\times3} \times B_{3\times5}$, $A_{5\times3} \times B_{3\times5} + \alpha C_{5\times5}$ and $A_{4\times4} \times B_{4\times4}$. The 4×4 matrices are benchmarked in place of 3×3 matrices since a translation and rotation operation can be combined in this way. We implemented these operations using various vector math libraries such as CLHEP, the Intel Math Kernel Library(MKL) [7], the S-Matrix library [8] and the Eigen [9] library. The speedup ratio with respect to CLHEP for each library is shown in figure 4. Intel MKL was between 5-20% slower than CLHEP for small matrix operations. This is probably due to its tuning for rather large sized matrix operations. Nevertheless, for $A_{5\times3} \times B_{3\times5} + \alpha C_{5\times5}$ it was about 2 times faster than CLHEP. The S-Matrix library, which is included in ROOT framework and based on C++ expression templates, provided a speed up between 3.2 to 4.4 times compared to CLHEP. The best speedup is achieved by Eigen library which achieved 6 to 12 times speed up with respect to CLHEP depending on the operation. Eigen is similar to S-Matrix in that it uses expression templates, however, on top of that, it also contains explicit vectorization with SIMD instructions,

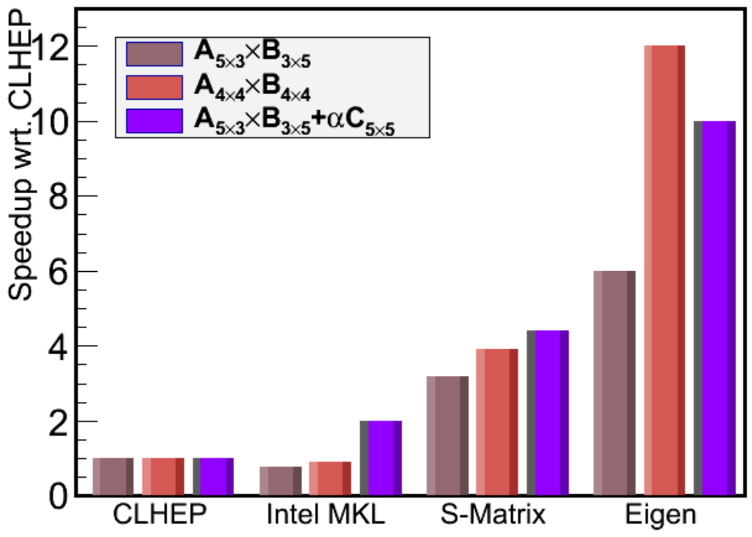


Figure 4. Speedup ratios of various libraries for different operations with respect to CLHEP library. The matrix sizes were found by instrumenting Athena and correspond to most commonly used matrix sizes in ATLAS reconstruction. The Eigen library performs significantly better than the other candidates.

avoids dynamic memory allocations and unrolls loops whenever it makes sense for fixed size matrices. Thanks to its significant speedup with respect to CLHEP, Eigen was choosen as a replacement for CLHEP. Migration from CLHEP to Eigen library in Athena framework is in progress.

5. Conclusions

The optimization of many-developer large code bases such as Athena is a challenging task. Using GOoDA we identified several performance problems and focused our efforts on these points. Replacement of magnetic field code alone provided up to 20% improvement in overall runtime while maintaining same physics performance and being used as the default field implementation in recent Athena releases. Using Pin we analyzed the most frequently used vector and matrix operations in Athena and better constrained the requirements of the vector math library. Using this information we picked the Eigen library as a replacement for CLHEP.

Both GOoDA and Pin are proven to be valuable tools for analyzing and improving Athena framework. Much information can be gained with little effort. Both tools can be used by average programmer at a basic level after some study, however it requires a certain level of expertise to fully exploit either tool.

References

[1] Evans L and Bryant P 2008 *JINST* **3** S08001
[2] Aad G *et al.* (ATLAS Collaboration) 2008 *JINST* **3** S08003
[3] Clemencic M, Degaudenzi H, Mato P, Binet S, Lavrijsen W *et al.* 2010 *J.Phys.Conf.Ser.* **219** 042006
[4] Luk C, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi V J and Hazelwood K 2005 *Programming Language Design and Implementation (PLDI)* 190–200
[5] Calafiura P, Eranian S, Levinthal D, Kama S and Vitillo R 2012 *J.Phys.Conf.Ser.* **396** 052072
[6] *CLHEP Web Page* URL `http://wwwasd.web.cern.ch/wwwasd/lhc++/clhep/index.html`
[7] *Intel Math Kernel Library* URL `http://software.intel.com/en-us/intel-mkl`
[8] *ROOT Manual* URL `http://root.cern.ch/root/html/MATH_SMATRIX_Index.html`

[9] *Eigen Web Page* URL `http://eigen.tuxfamily.org/index.php`