Performance evaluation and optimization of Geant4 on GPUs

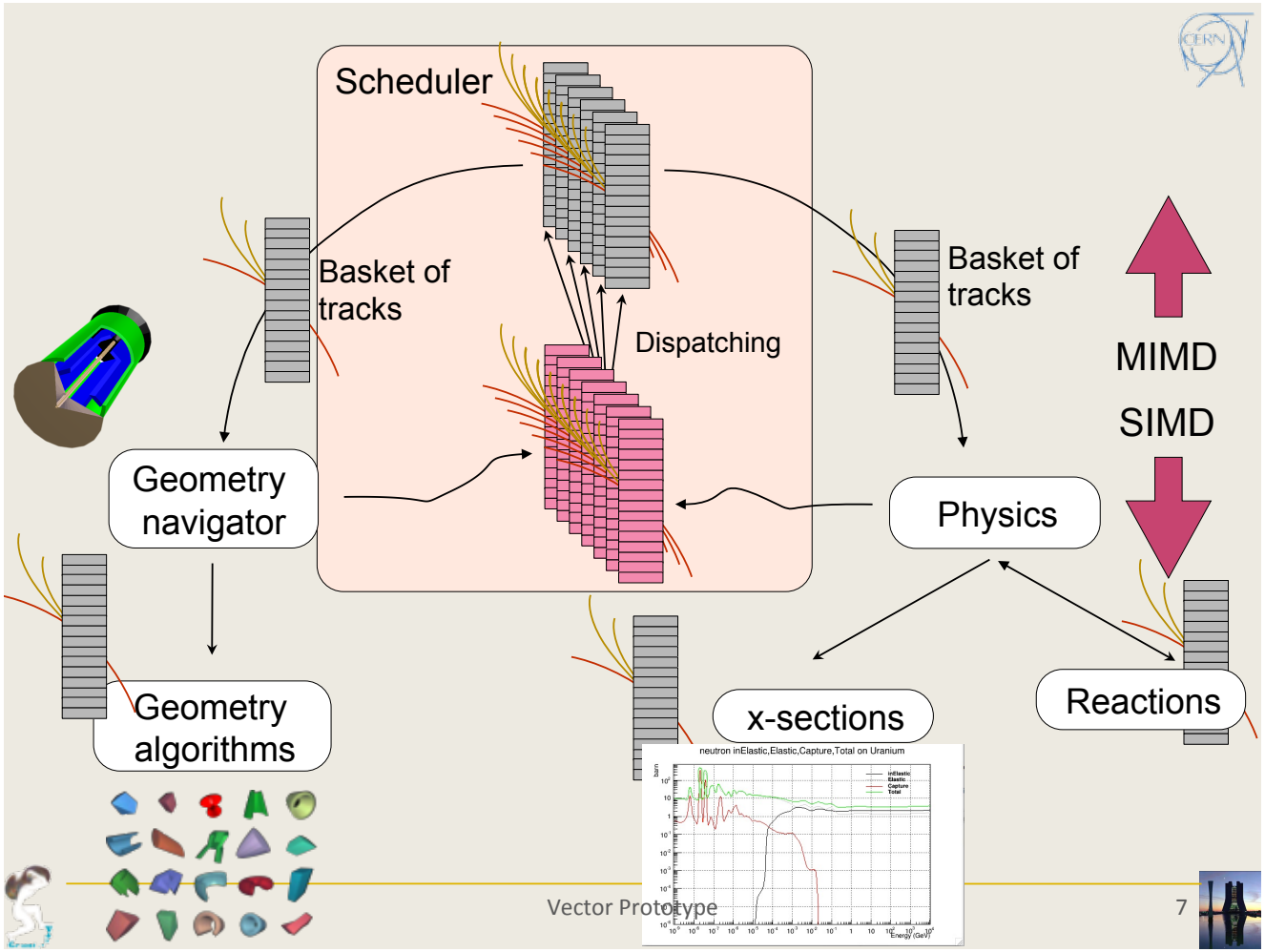
Azamat Mametjanov
LANS Performance Group
Mathematics and Computer Science Division
Argonne National Laboratory

*Collaboration between US DOE **HEP** Geant4 Reengineering and*
*US DOE SciDAC institute **SUPER**: Sustained Performance, Energy and Resilience*

Geant4

- Geant4 is open-source software package for accurate simulation of particles passing through matter with tools for:
  - Geometry of the system
  - Properties and composition of materials
  - Properties of fundamental particles: neutrons, protons, ions, hadrons
  - Physics of interaction of beam particles with detector matter
  - Tracking and detection of collision events
  - Capture, visualization and analysis of particle tracks and events
- Applications are built using Geant4 tools by extending/adding new physics and time-stepping of particle interactions
- Geant4 is a C++ object-oriented framework
  - Type-parametric geometry coordinates: e.g. float or double scalar, Vc::Vector<Scalar>, CilkVector<Scalar>, cudaTextureType<Scalar>

Reengineering of Geant4



Philippe Canal: "Simulation Vector Prototype", AHM on Feb 5 2014

Motivation

- Geant4 enables high-precision particle tracking through space
  - Relatively high memory intensity
- Recent advances in computer architecture
  - Multi-core CPUs: 4—32 cores
    - Deeper SIMD/vectorization units (SSE, AVX, AVX2): 2/4/8-wide DP flops
  - Many-core accelerators (GPUs & MICs): 256—512 lightweight cores
  - Multi-node clusters:
    - X chips/blade, Y blades/rack, Z racks ➔ X*Y*Z CPUs
- Simple up-scaling is energy-inefficient
- Need to re-engineer for efficiency
  - Improve vectorization for higher bandwidth
  - Increase concurrency for lower latency

Objective

- Is the code executing at peak rate?
  - What is the rate-limiting factor: flops or bytes?
- Are there any stalls?
  - Cycles with no instruction issue
- Contributions
  - Profile benchmarks
  - Identify inefficiencies in the library
  - Improve

Methodology

- Compiler is a black-box
  - May need to read its output to determine how well it optimized the sources
    - Assembly '*.s' files for C/C++
    - Program listing '*.lst' files for Fortran
  - Different compilers have varying strengths
    - Platform-oriented: IBM, Cray, Intel, NVidia
    - Language-oriented: NAG, PGI
    - Customizable/Extensible: GCC, Clang
- Run-time profiling of benchmarks that use Geant4 API
  - Collect hardware performance counters: cycles, instructions, events
  - Calculate metrics to identify inefficiencies

Profiling environment

- Hardware
  - Intel Xeon E5-2620 Sandy Bridge with AVX
    - 6 cores @2.0 GHz
      - Peak of 6 x 8 DP-flop/cycle: 96 Gflop/s
    - 32 GB DDR3-1333MHz @42.6 GB/s
      - 15 MB L3$, 6x256 KB L2$, 6x32 KB L1I$ & L1D$
  - NVidia Tesla K20m Kepler
    - 13 SMs with (192 SP + 64 DP + 32 SF) core/SM @706 MHz
      - Peak of 13x64 x 2 DP-flop/cycle: 1175 Gflop/s
    - 5 GB GDDR5-2.6GHz @208 GB/s
      - 1.5 MB L2$, 13x16 KB L1$
- Software
  - Linux x86_64 Scientific Fermi v6.3 (Ramsey)
  - GNU GCC compiler v4.8.2
  - NVidia CUDA SDK v7.0

	CPU	GPU	GPU/CPU
GFlop/s	96	1175	12.24
GByte/s	42.6	208	4.88

Vectorized Geometry

- TubeBenchmark
  - Performs particle geometry transformation and checks
    - Inside
      - The volume fully contains the new particle location
      - The new location is on volume surface
    - To
      - Distance to volume
      - Safety to volume
    - Out
      - Distance out of volume
      - Safety out of volume
  - Six micro-benchmarks

Vectorized Geometry

- Geometry is 3-dimensional with double-precision coordinates
  - Benchmark checks AOS vs. SOA coordinate storage
    - Vc library is used as a backend to convert SOA access pattern into SIMD SSE or AVX instructions
- Geometry is also parameterized on volume shape types
  - Checking the shape at run-time leads to many conditional instructions
  - C++ templates and static type specialization dispatches to appropriate volume at compile-time
    - E.g.: Volume → Tube → Full or Hollow Tube → Half or other_phi Tube
- Geometry objects can also be constructed in GPU memory and C++ code is compiled into CUDA kernels
- All implementation versions are compared against existing baseline geometry implementation using ROOT library API

Tube Benchmark Baseline

./TubeBenchmark -npoints 1024 -nrep 1024
-rmin 0 -rmax 5 -dz 10 -sphi 0 -dphi 6.28

	Inside	Contains	In/Con	DistToIn	SafetyToIn	DisIn/SafIn	DistToOut	SafetyToOut	DisO/SafO
ROOT	n/a	0.132269s	n/a	0.227153s	0.067418s	3.37	0.177144s	0.556240s	0.32
Unspecial	0.043689s	0.031367s	1.39	0.178060s	0.036499s	4.88	0.152497s	0.070306s	2.17
Specialized	0.042583s	0.029720s	1.43	0.164871s	0.035811s	4.60	0.152591s	0.070230s	2.17
Vectorized	0.024635s	0.014892s	1.65	0.105517s	0.027273s	3.87	0.060822s	0.023602s	2.58
CUDA	0.005418s	0.005415s	1.00	0.006128s	0.005331s	1.15	0.005452s	0.007886s	0.69

- Statically specialized version is faster than non-specialized version
- SIMD version is faster than all CPU-based versions
- CUDA version is faster than SIMD version by 3x—10x

Vectorized Geometry on CPU

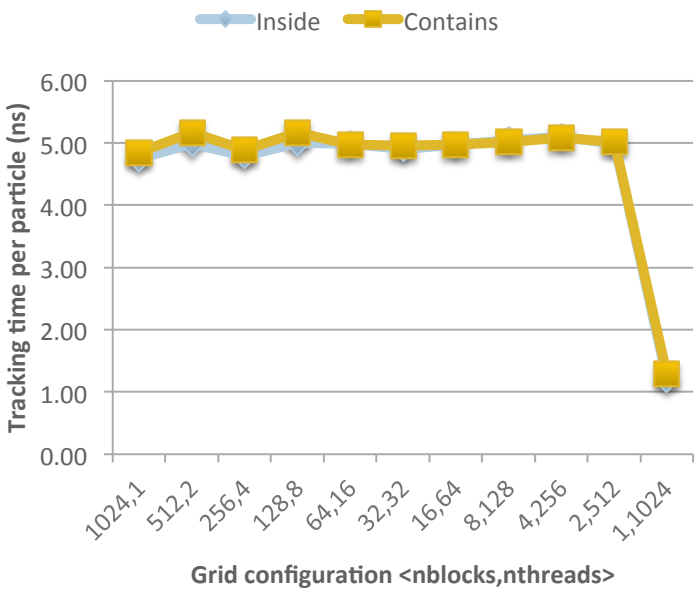
- PAPI_SP_OPS is 4 (essentially 0) ⇒ there are no SP ops: Good
- PAPI_VEC_SP is 0 ⇒ there are no SP vector instructions: Good
- For RunInsideVectorized:
  - PAPI_TOT_INS: 1.23e+8 (0.8 ipc)
  - PAPI_DP_OPS: 4.65e+7 ⇒ 4.65/12.3 = 37.8% of total instructions
  - PAPI_VEC_DP: 4.65e+7
  - PAPI_TOT_CYC: 1.54e+8 ⇒ 4.65/15.4 = 0.30 flop/cycle ⇒ 3.75% of peak
- For RunToInVectorized:
  - PAPI_TOT_INS: 2.58e+8 (0.7 ipc)
  - PAPI_DP_OPS: 1.51e+8 ⇒ 1.51/2.58 = 58.53% of total instructions
  - PAPI_VEC_DP: 1.51e+8
  - PAPI_TOT_CYC: 3.70e+8 ⇒ 1.51/3.70 = 0.41 flop/cycle ⇒ 5.13% of peak
- For RunToOutVectorized:
  - PAPI_TOT_INS: 1.34e+8 (0.6 ipc)
  - PAPI_DP_OPS: 9.03e+7 ⇒ 9.03/13.4 = 67.39% of total instructions
  - PAPI_VEC_DP: 9.03e+7
  - PAPI_TOT_CYC: 2.12e+8 ⇒ 9.03/21.2 = 0.43 flop/cycle ⇒ 5.32% of peak
- All DP ops are vector instructions ⇒ Vectorization is complete!
- Low arithmetic intensity ⇒ Explore prefetching…

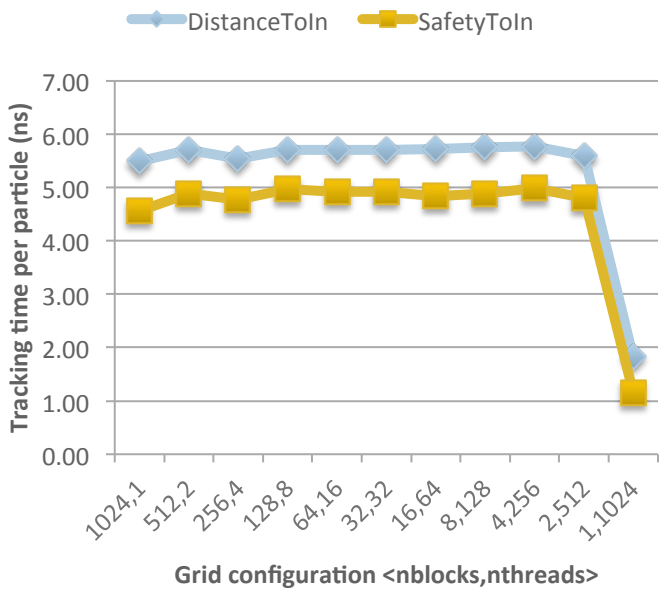
Peak: 8 DP Flop/cycle

Vectorized Geometry on GPU

- Threads are launched over points
- Number of threads per block is fixed at 256
  - Number of blocks is npoints / 256
- What are the effects of various thread grid combinations?
  - Maximum number of threads per block is (1024, 1024, 64)
  - Maximum number of blocks in a grid is (2^{31}, 2^{16}, 2^{16})

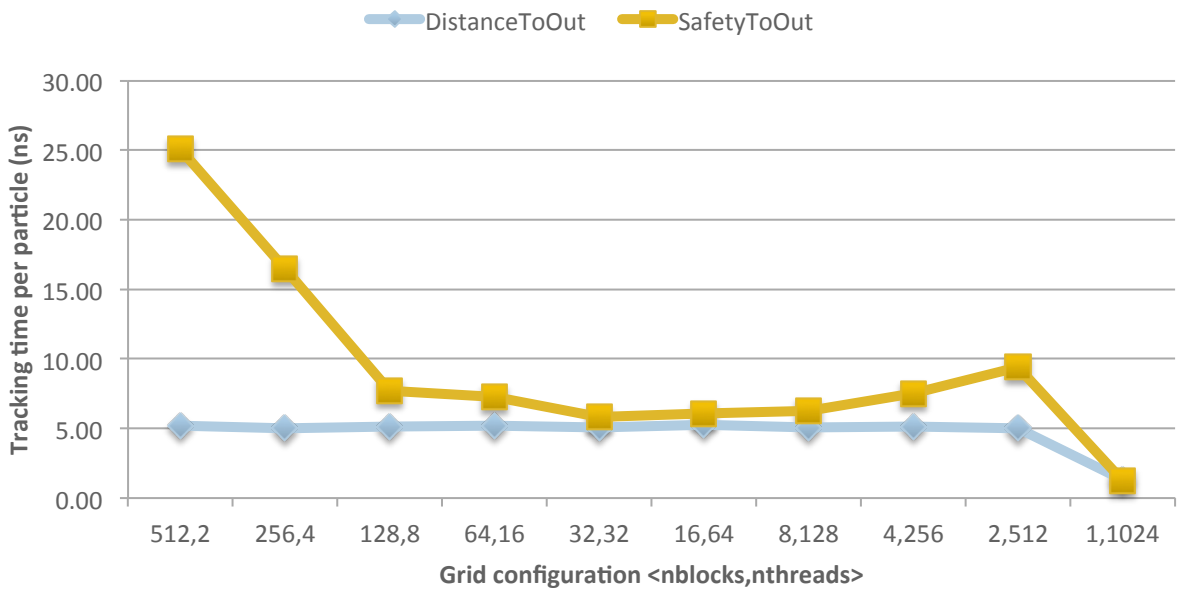
Vectorized Geometry on GPU





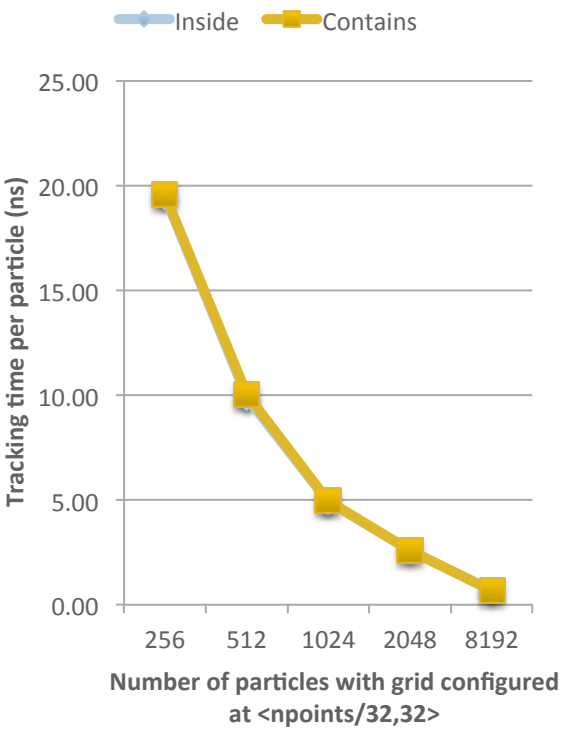
- Tracking time per particle (ns) is stable across all configurations
- Launching 1024 threads/block leads to out-of-memory errors
  - Benchmarks need to catch this type of errors
- Tracking time stability indicates memory as the limiter

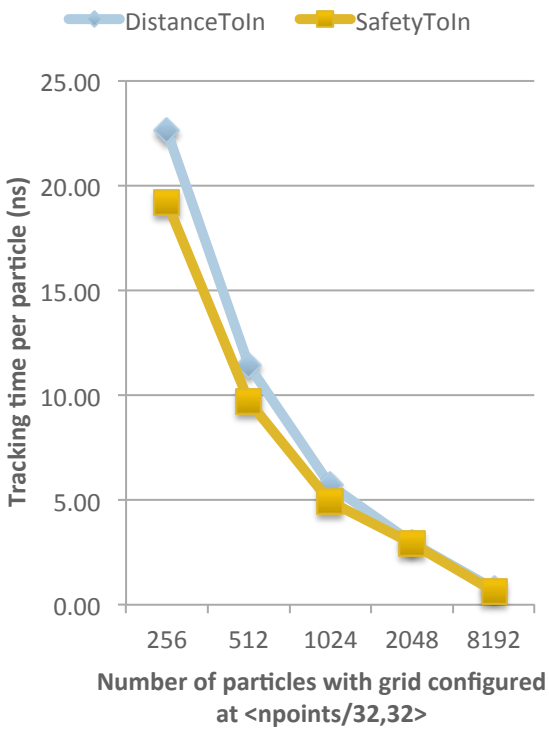
Vectorized Geometry on GPU

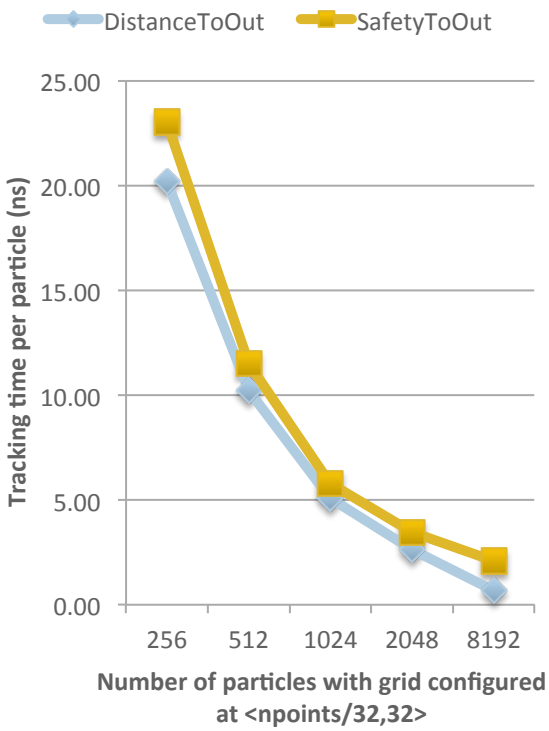


- Best tracking time is obtained at <32,32> configuration
  - Minimal scheduled block size is a warp of 32 threads
- Smaller blocks show computation as the performance limiter
- Larger blocks lead to cache-thrashing

Vectorized Geometry on GPU







- Increasing the problem size leads to lower tracking time per particle
  - Scaling is less than linear: memory bandwidth saturation

Vectorized Geometry on GPU

	Instructions/cycle
ContainsBenchmarkCudaKernel	0.22
InsideBenchmarkCudaKernel	0.22
DistanceToInBenchmarkCudaKernel	0.32
SafetyToInBenchmarkCudaKernel	0.29
DistanceToOutBenchmarkCudaKernel	0.36
SafetyToOutBenchmarkCudaKernel	0.30

- Collection of flop counts fails with CUDA 7.0, 6.5, 6.0 for any number of particles: seg-fault or profiler error
  - Efficiency in terms of % of peak unavailable

- Arithmetic Function Unit Utilization is “low” for all kernels: memory saturation

Future work

- For each program unit: class, function, loop, block
  - Calculate flops & bytes
    - HPCToolkit, TAU, …
- Identify inefficiently executing code
  - Memory: low achieved bandwidth
  - Compute: instruction/cycle relative to peak
- Code refactoring
  - Memory: prefetch, loop transformations, …
  - Compute: scale-out to CUDA/OpenACC, OpenMP, MPI

Conclusion

- Profile benchmarks
  - CPU:
    - IPC: 0.6-0.8, Efficiency: 3.7-5.3 % of peak
  - GPU:
    - IPC: 0.22-0.36, Efficiency: low
- Identify inefficiencies
  - CPU:
    - Insufficient parallelization
  - GPU:
    - Number of threads/block is suboptimal. Tuning suggests 32 threads/block.
- Improve
  - CPU:
    - Multi-threading with parallelization over particles
  - CPU & GPU:
    - Prefetching