Impact of different compilers and build types on Geant4 simulation execution time

CATERINA MARCON - LUND UNIVERSITY
CHEP 2019 – 07.11.2019
Motivation

• Currently, Monte Carlo detector simulation at LHC can occupy up to 40% of World Wide LHC computing grid’s resources. This percentage is set to grow when LHC luminosity will be further increased.

• It is necessary to find a new approach for improving the execution time of simulations without sacrificing the quality of simulated data.

• The purpose of this preliminary study is to investigate how to reduce the Geant4 simulation execution time.

• This is achieved by running standalone Geant4 simulations, whose performance can then be evaluated independently from other libraries and control frameworks.
Method

Several factors can have an impact on the compilation process:

• **Static linking** is expected to lead to a **faster execution**, and will be compared here to the traditional dynamic linking.

• Compiler **optimization**. Machine code can be optimized by:
  1. avoiding redundancy (reuse instead of recompute data);
  2. reducing amount of code to fit as much as possible into CPU cache;
  3. preferring sequential code instead of many jumps, parallelizing as much as possible (e.g. loops), etc.

• Compiler **version**.
Method

- As a benchmark, **standalone G4 simulation** with two different geometries (from A. Dotti [1]) has been used. **50 GeV pions** are used as source particles. The number of simulated primaries varies according to the detector geometry.

- Compiled G4 (version 10.5) both **statically** and **dynamically**.

- Three versions of the GCC compiler, namely **4.8.5, 6.2.0 and 8.2.0**, have been used for these investigations.

- A comparison between four GCC optimization levels (**Os, O1, O2 and O3**) have also been performed. The default level used by most build systems is **-O2** and it will be used as reference.

- The computations were carried out on a **standalone machine at CERN IT** and on a **university cluster** in Lund.

- CPU and memory resources on both machines (standalone and cluster) were **exclusively allocated** to the simulations and not shared with any concurrent process other than the minimum OS tasks.

Computing resources

CERN standalone machine

- CPU: Intel Xeon E5-2630 v3 2.40GHz
- 16 cores / 32 threads
- 20 MB Cache (L1: 64 KB, L2: 256 KB, L3: 20 MB)
- 64 GB RAM
- Filesystem: XFS
- Operating System: CentOS 7

Compute node on Lund University cluster

- CPU: Intel Xeon E5-2650 v3 2.30GHz
- 10 cores / 20 threads
- 25 MB Cache (L1: 64 KB, L2: 256 KB, L3: 25 MB)
- 128 GB RAM
- Filesystem: IBM General Parallel File System (GPFS)
- Operating System: CentOS 7
The computations were carried out on CERN machine considering 5000 initial events and using 4 threads. The computation was repeated 3 times for each configuration.

The static approach, for all the GCC versions, reduces the execution time by more than 10% in some cases.

Regardless of the build approach, switching from GCC 4.8.5 to GCC 6.2.0 and GCC 8.2.0 results in an average of 30% improvement in the execution time.

A static build with GCC 8.2.0 leads to an improvement of almost 34% with respect to the default configuration (GCC 4.8.5, dynamic, O2).

The different GCC optimizations do not seem to have visible effects on the execution time.
The computations were performed on the university cluster considering 5000 initial events and using 4 threads. The computation was repeated 5 times for each configuration.

- The static approach allows a performance gain: it reduces the execution time by more than 10% in some cases.

- The impact of different compilers is not relevant as in the previous case.
The computations were carried out on CERN machine considering 50000 initial events and using 4 threads. The computation was repeated 3 times for each configuration.

- The static approach, for all the GCC versions, reduces the execution time by more than 9% in some cases.
- The impact of different compilers is not relevant as in the full geometry case.
- The different GCC optimizations do not seem to have visible effects on the execution time.
The computations were performed on the university cluster considering 50000 initial events and using 4 threads. The computation was repeated 5 times for each configuration.

The static approach allows a performance gain: it reduces the execution time by more than 10% in some cases.
The number of events per thread has been set to 1250 events for each configuration. GCC 8.2.0 has been used.

The improvement between static and dynamic linking is confirmed in all cases on both machines (standalone and cluster).
The number of events per thread has been set to 12500 events for each configuration. GCC 8.2.0 has been used.

The improvement between static and dynamic linking is confirmed in all cases on both machines (standalone and cluster).
Future steps

- Perform measurements with other detector configurations (in progress).

- Perform measurements with different physics (consider different generated particles).

- Consider different compilers beyond GCC:
  1. clang (in progress);
  2. icc (intel);
  3. pgf (portland).

- Consider more advanced methods of compile-time optimization (e.g. link-time optimization (LTO)).

- GDML-based geometries, used for these studies, are not compatible with all the detector components. We need to adapt our benchmark simulation to support newer geometry definitions.
Conclusions

• Execution time for simulations based on Geant4 can be significantly improved by changing the default build method: linking Geant4 and its associated libraries statically can produce binaries that run even 10% faster.

• Switching from gcc 4.8.5 to 8.2.0 results in a reduction of the execution time up to 25%.

• Static libraries are embedded into the executable, resulting in a much larger size (~700 MB) than the corresponding dynamically-linked code (~ 2.5 MB).

• The different GCC optimizations do not seem to have visible effects on the execution time.
Thank you for your attention
## Optimization levels [Backup]

<table>
<thead>
<tr>
<th>option</th>
<th>optimization level</th>
<th>execution time</th>
<th>code size</th>
<th>memory usage</th>
<th>compile time</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>optimization for compilation time (default)</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-O1 or</td>
<td>optimization for code size and execution time</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>-O2</td>
<td>optimization more for code size and execution</td>
<td>--</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>-O3</td>
<td>optimization more for code size and execution</td>
<td>---</td>
<td>+</td>
<td>+++</td>
<td>+++</td>
</tr>
<tr>
<td>-Os</td>
<td>optimization for code size</td>
<td>--</td>
<td>++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Ofast</td>
<td>O3 with fast none accurate math calculations</td>
<td>---</td>
<td>+</td>
<td>+++</td>
<td></td>
</tr>
</tbody>
</table>

+ increase  ++ increase more  +++ increase even more  - reduce  -- reduce more  --- reduce even more