CMS-SIM PROFILING

Alejandro Alonso, Josep Flix, and Juan Carlos Moure

Abstract—CERN produces millions of images from the collisions in the accelerator. This images must be cleaned up to extract important data. PIC is one of the organizations that processes the images.

It is expected that the amount of data to process increases on a 7x factor on the next few years. However, the budget for the organizations doesn't increase each year, so the hardware can only be updated with better hardware that the market offers with the same price as the hardware acquired years before. This update is not able to accomplish the expected 7x factor, so we need to find ways to optimize money, resources and applications.

1 FIRST LOOK AT CMS-SIM

CMS-SIM is the part of the program that simulates montecarlo events using a sql file with data of the collider sensors and structure. It has the following elements:

Input: SQLite file with collider detector data, the file is only some MB, it is located on repositories in CERN and USA. But we make use of a squid cache to access this data more efficiently, so the actual input comes from PIC squid server.

Event compute: Main operation in the execution, it's the part of the program that generates one image that represents 1 collision of two particles. Other versions of cms-sim, like "gen-sim-digi-reco", add more particle collisions on each event, which is closer to the real images that we get from particle collision.

Output: .root file with the "ttbar" simulations. The size of the file depends on the number of events, it's size is (nEvents * sizeof(image) + 1MB) where nEvents is the number of events.

Structure: Analyzing logs and profiling graphs we can extract 3 parts:

Initialization: This is the sequential part where the needed plugins are loaded and also the SQLite file is loaded to memory. Apparently this part can be runned in 2 threads but not more. Also this part time may vary, since we depend on the server we are accessing to download the file.

Core event computing: This is the main part of the sim where the simulation compute happens. This part can be parallelized to a maximum number of threads, since each thread will compute 1 event at a time. So at the beginning of the event, the thread reads data from the SQL file located in memory and computes the event, once it's finished it will write the results on memory and they're only written back to disk once the buffer is full.

Termination: The program closes all threads and writes back to disk some of the remaining event data. Notice that resilient event data on this face is not really big since we're writing the data periodically on the second part. We will see this behaviour on the I/O graphs.

2 PIC TEST ENVIRONMENT

To profile cms-sim, we have isolated one of PIC's nodes. This node has 2 sockets of 8 cores with hyperthreading each. That makes a total of 32 possible threads in 16 cores. The actual processors are "Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz" which have a intel Haswell architecture.
The memory is 64GB and disk is about some teras.
To execute the code we use the CVMFS, the shared file system that allows us to run cms-sim without loading it from repositories everytime that we need to run it.
The cms-sim version that is used for the analysis is CMSSW_10_2_9.

To profile cms-sim, we used 3 different profilers:
PrMon [3]
Trident [4]: Trident can only be runned in intel Haswell architecture due to the use of specific hardware counters.
Linux Perf [5]

3 NUMBER OF THREADS AND EVENTS ANALYSIS

To start profiling, we executed cms-sim with different threads and number of events. The numbers where the following: number of threads as nthreads (1,8,16,32), number of events as nEvents (128,512,1024,2048,4096).
To judge performance we will use the number of events

per second, since computing events is the main goal of cms-sim. So output divided per amount of time is what really matters.

We will now discuss the differences between different number of events and threads executions with the results of PrMon [3]:

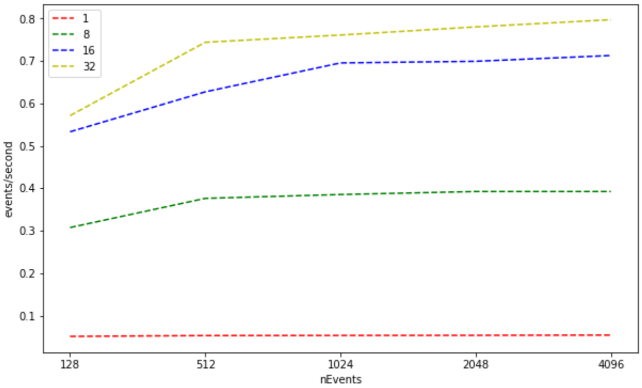

Figure 1 . Events per second of CMSSW_10_2_9 with different number of threads and events.

Speedup	128 events	512 events	1024 events	2048 events	4096 events
8 threads	6.00	7.08	7.21	7.30	7.23
16 threads	10.40	11.80	13.01	13.01	13.13
32 threads	11.14	14.00	14.23	14.52	14.69

Table 1 . Speedup based on sequential execution.

Looking at Figure 1 judgement will be that a threaded execution is a decent optimization since it almost doubles the performance when doubling the threads. Hyperthreading doesn't usually help, so it is common to see that 32 threads isn't much better than 16, but in this scenario it gives a little more performance. We will see with further analysis why 32 threads gives this improvement.

You can notice at Table 1 that the higher the number of events, the better is the performance, for example a 32 thread version gets only a 11.14 speedup with 128 events but it reaches 14 when increasing the number of events. Also the higher is the number of threads it requires a higher number of events to get the maximum performance, for example, the 8 thread version has less difference in performance between the least number of events and the maximum.

Having better performance with a higher number of events is due to the "initialization" part mentioned in the previous section, which means that a certain number of events makes the "core event computing" part big enough to make "initialization" part insignificant so the threads will be dividing the main work of the program.

In Figure 1 we can see how each number of threads behaves with different number of events. For all number of threads 1024 events seems to be a stable number, since it doesn't increase too much with a higher number. So a 1024 event execution is a valid execution to judge from the point of resource usage.

4 Resource usage analysis

We could use PrMon [3] for a resource usage analysis, but since we are on a Haswell architecture we can use a more accurate tool named Trident [4] that gives us much more information than PrMon [3]. The only problem is that Trident only allows us to run executions that fully load the processors of the machine including hyperthreading, this means we must run 32 threads executions.

The following graph corresponds to a raw cms-sim execution with 1024 events, 32 threads and 1 process:

4.1 CPU efficiency

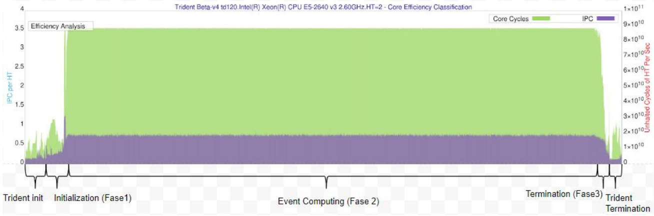

Figure 2 . IPC of each HyperThread and cycles assigned on CPU to that thread.

Figure 2 shows at the bottom graph, for each hyperthreaded-core, how many instructions per cycle (IPC) the thread achieves. Note that each core has 2 hyperthreads, so the actual core IPC is double the amount shown. That gives us in a 0,8 * 2 = 1,6 IPC vs 4 IPC as the maximum the computer can achieve.

You could say that it is pretty inefficient due to the IPC being 2,4 cycles lower than the actual maximum, but actually these are decent performance results since it is almost impossible to get the maximum IPC due to several reasons. Also a lot of instructions are divided into micro operations, which means that our processor is doing more work, since the maximum IPC doesn't consider that.

Figure 2 green colour shows us the amount of core cycles assigned to that thread, this only gives us the information that the thread is almost getting all the core cycles posible which means the scheduler is giving the 100% of the CPU. This helps us see the different phases of the application mentioned on section 2. As you can see during initialization and termination the core cycles assigned are lower than the one assigned on the event computing.

4.2 Processor pipeline port usage

Each cpu core has different ports where the micro instructions are placed, each port can process a different type of instruction and each type of instruction has a duration in cycles different from each other.

Taking a look at the Haswell port distribution on Figure

3, which shows what instructions are being executed on each port and will tell us how the program works at assembler instruction level:

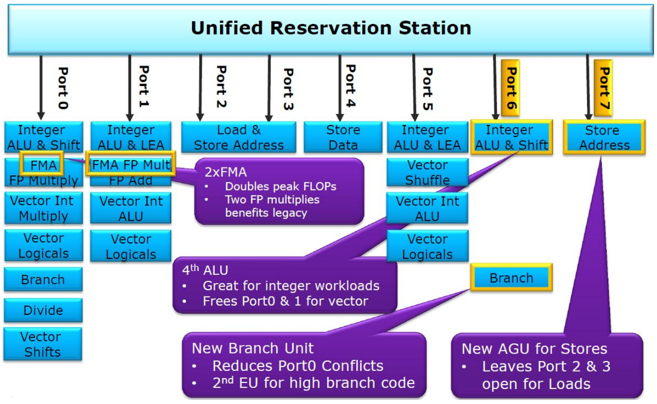

Figure 3 . Pipeline scheme of the processor used.

Figure 3 shows which micro instructions are managed on each port. Ports 0,1,5,6 are associated to compute instructions and 2,3,4,7 to memory instructions.

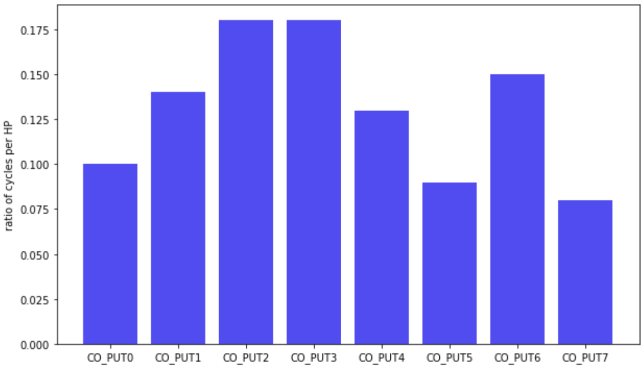

Figure 4 . Use percentage of each pipeline port for each Hyper Thread

Figure 4 shows the percentage of cycles per second we spend on each processor pipeline port. The ratio of cycles per HP determines how many cycles the port was busy.
The most used resources are ports 2,3 which make memory operations, those reach a 17,5% of use, if we calculate the core usage it duplicates due to using 2 threads on each one. This means we have a maximum usage of 35%, even though results are taken from a fraction of time and this usage can be higher at some point, the program is pretty stable and 35% is a very low resource usage, so we could almost confirm that there are no CPU resources saturated.
As CPU is not a problem, let's go on and check memory and disk.

5 MEMORY HIERARCHY AND DISK ANALYSIS

Trident [4] and PrMon [3] have several features but neither of that tools can measure cache misses, that's why Linux Perf [5] is used to measure that.

5.1 Cache misses

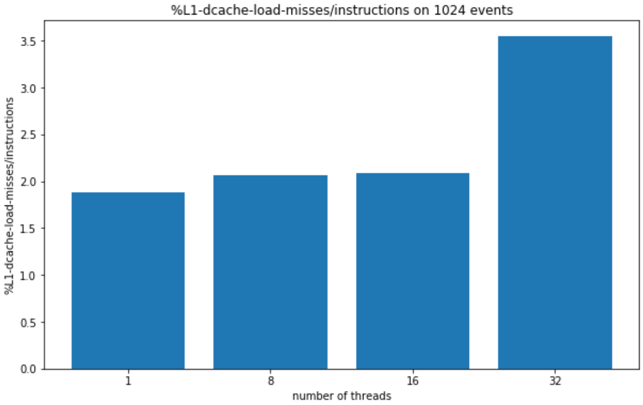

Figure 5 . Percentage of load access to L1 that result on miss in comparison to the program instructions.

L1 is the closest cache to CPU, so the fastest to access, and there is one for each core, so it is shared only by the 2 core hyperthreads. That's the reason why looking at Figure 5 we see that the 32 thread version has x1,75 more misses, almost double the misses. This is because 2 threads are trying to load information from the same cache, which is causing more collisions, and the other versions only 1 thread loads from cache so it doesn't collide with another thread accesses.

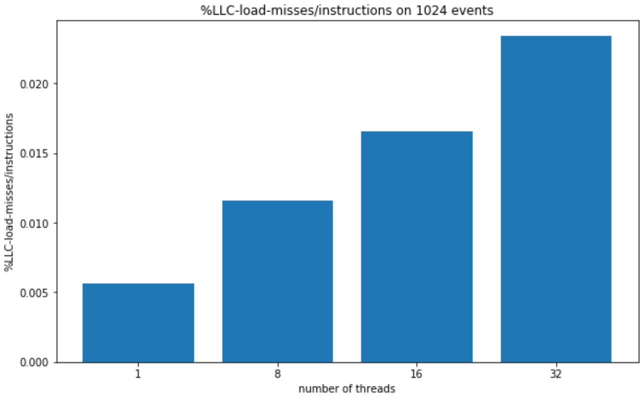

Figure 6 . Percentage of load access to LLC that result on miss in comparison to the program instructions.

Figure 6 shows the ratio of misses in LLC (Last Level Cache), which is the closest cache to memory, so the slowest cache to access but still faster than memory access. This cache is shared by all cores, so the higher the number of threads the higher the number of accesses.

This only shows how the cache misses scale with the number of threads compared to the total instructions, but if you take a look at the number of cache accesses compared to the number of misses, you will see that the cache misses are between a 6-10% on L1 and 2-4% on LLC. This added to the fact that memory access is very low on cms-sim, makes cache misses almost irrelevant on this executions.

Also, note that when using 32 threads we get more cache misses, but actually thanks to hyperthreading, when one thread wastes time with a cache miss, the other thread will make use of the CPU, so we aren't wasting any time with cache misses and memory accés on 32 threads. This is the reason why 32 threads is slightly better than 16.

Let's move again to Trident [4] to check the main memory behaviour and see if there's something wrong:

5.2 Memory accesses

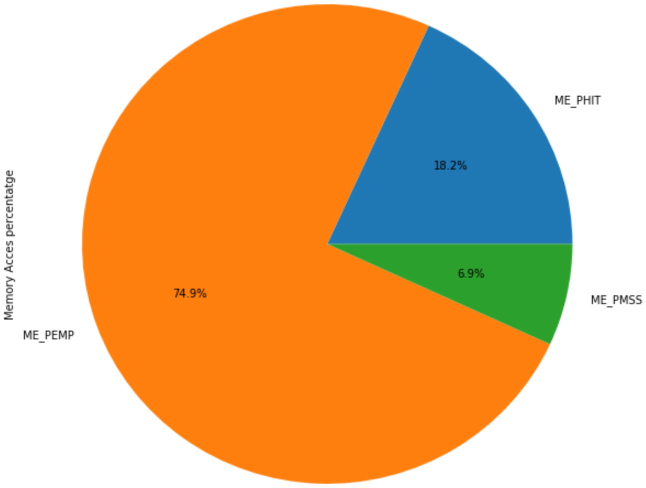

Figure 7 . Percentage of each main memory access type, each type explained below:
# ME_PHIT - Memory transaction resulting in page hit
# ME_PEMP - Memory transaction resulting in page empty
# ME_PMSS - Memory transaction resulting in page miss

In Figure 7 one can see that almost all the access that are made result on a Page empty, this is actually pretty bad. Page Empty access takes double the time than a Page Hit and a Page Miss takes double the time of a Page Empty. These results are due to inefficient memory access, data from the memory is being accessed in a very sparse way.

This would worsen the performance if it wasn't by the fact that the number of memory bytes accessed are pretty low as we can see on Figure 8. Note that this behaviour would affect the program if in some way we raise the amount of memory access. This can be done by raising the number of threads, which we can't see due to our computer limits, or by rising the number of processes.

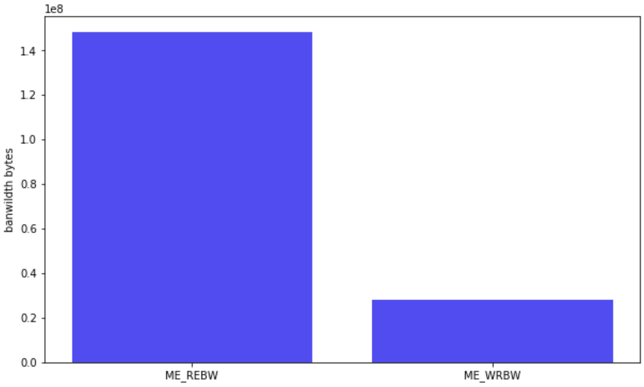

Figure 8 . Number of bytes read and written to main memory.
# ME_REBW - Memory read bandwidth
# ME_WRBW - Memory write bandwidth

5.3 IO Access

Figure 9 shows on the bottom how many times per second do we access IO and on the top how many bytes are transferred with each access, on the right the % of usage is displayed, showing how much impact have the numbers related to the system.

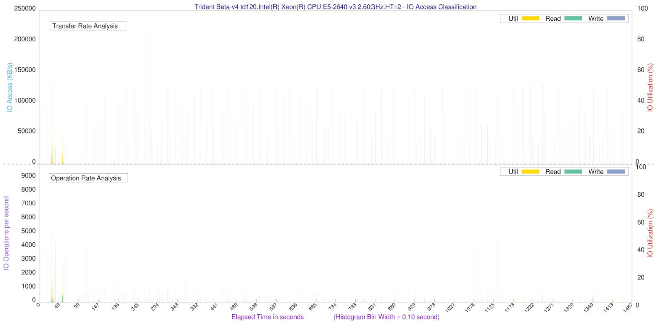

Figure 9 . Disk bytes transferred and number of operations.

As one can see the IO is very sparse and doesn't really take an impact on the system if one takes a look at the %. Some of these IO accesses may reach 80% bandwidth utilization but it's not enough to affect the system in any way.
Having low amount of IO also tells us that all the data we're accessing is located into memory since we don't have to load it from disk during the execution.

6 PROCESS DISTRIBUTION EXPERIMENT

To test if the number of memory accesses affect the program, we made a test executing more than one process instead of using the maximum number of threads with one process. This has been done with the docker version of this program, which is not different from the normal version on a single process execution.

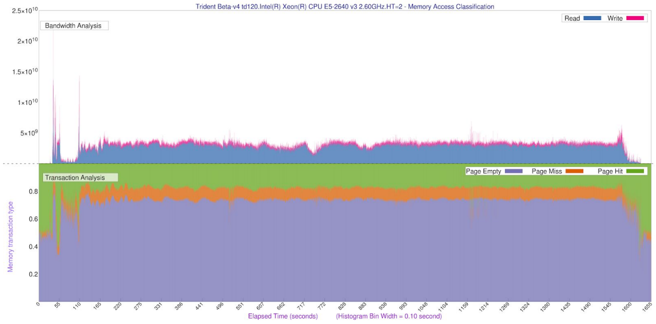

Figure 10 . Main memory bytes transferred and type of memory access percentage.

Running 32 sequential processes instead of 1 process of 32 threads, means that we do 32 times the initialization part, so we are loading 32 SQL files into the current memory, these files will be accessed by 32 different processes which means the memory bus will have to access even more sparse data on the memory and the amount of data to be accessed will be bigger.

Looking at Figure 10 we see that this results only make the execution increase from 1500 seconds to 1650 seconds which is a 10% more time spend on execution, this is the worst case scenario for rising the number of processes. 4 processes or 8 processes give almost the same performance as 1 process.
This doesn't affect enough the performance to worry about it. Just be careful with the number of process used.

At last we will take a look at the IO of the program using Trident [4] again, but the program behavior so far shows that there won't be a IO problem due to the number of access.

7 Final analysis

We have seen that the program scales properly with the number of threads on a high number of events, and that the only problem that we have is some inefficient memory accesses. To confirm that the application behaviour scales properly we will use perf to see how IPC behaves on each thread since we can't use Trident [4] or PrMon [3] for this:

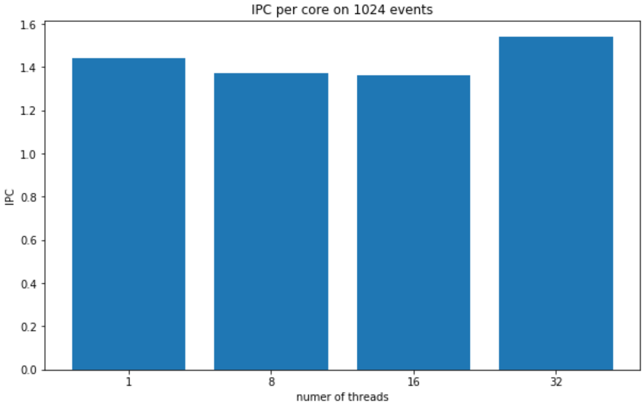

Figure 11 . Core IPC for CMSSW_10_2_9 with different number of threads and 1024 events.

As you can see on Figure 11, threading the program has almost no performance loss on each thread, compared to sequential execution. Note that 32 threads have double than 16 threads but more than half the IPC so it means that the thread performance is worst but the core performance is better.

8 Different CMS versions analysis

After doing a review of one of the newest versions of CMS (CMSSW_10_2_9), we have decided to compare it to other versions, not all are available since they were compiled in a different environment.

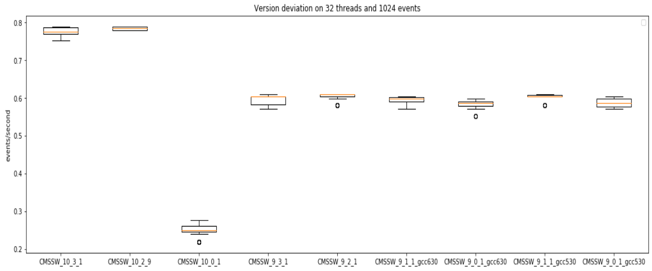

Figure 12 . Execution of several instances of each version under the same conditions to check deviation.

To ensure that all data collected from the versions is not arbitrary data, we have made several executions and checked if one executions can give different performance from other just by odds.
As you can see on Figure 12, the boxplots are pretty small, which means executions results are pretty close to each other. Executions can only give a number of events/second within the range of that box.

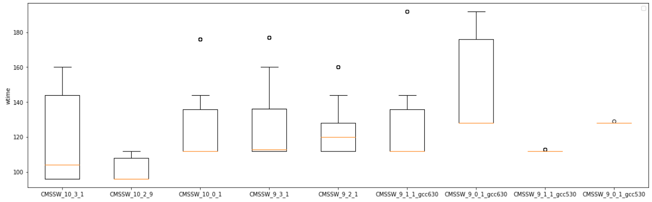

Figure 13 . Execution of several instances of each version with low amount of events to check initialization deviation.

Figure 13 is the same as Figure 12 but a low amount of events is used. This shows how the initialization part behaves, and as you can see the boxes are pretty big. This means that the duration of the initialization part is very arbitrary. So comparisons of the versions on low amount of events can give erroneous information.

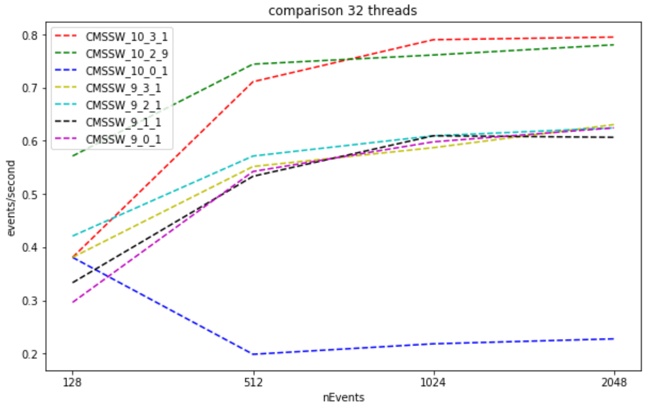

Figure 14 . Comparison of the amount of events per second that every version can do with a different number of events.

As we can see in Figure 14, we can group the versions analysed in 3 groups:

Group one are the versions of the 10th family that reach almost a 0.8 events/second on a high enough amount of events. These are the versions used on the main analysis of this project. Note that this versions use the gcc730 compiler.

Group 2 are the versions of the 9th family, that reach 0.6 events/second on the correct amount of events, all the versions follow almost the same pattern. Note that this versions use the gcc630 compiler, but versions 9_1_1 and 9_0_1 can be configured to use gcc530.

Group 3 is only version 10_0_1 which shows an extrange behaviour opposite to all versions and a really low events/second, this version should be reviewed to look for code problems. Note that this versions use the gcc630 compiler.

These 3 groups might be a hint to the compiler being a very important factor on the application performance, since the better versions use newer compilers and give a 25% more performance, which correlates with another investigation by Caterina Marcon [6], which explains that we can vary performance on a 25% more or less using different compilers on a different application used at CERN.

9 Conclusions

The scalability increasing the number of threads is pretty good, only if the number of events is big enough to make the initialization and termination part of the execution negligible. The proper number of events is about 100 events per thread.

CPU resources, E/S or memory bandwidth aren't a problem. Neither is memory capacity, since only 4% of the memory is used for storage of the sql file, that we need to compute the events, and also the generated event information, which is periodically written to disk.

We have seen that memory accesses can become a problem if the number of accesses increases. The number of accesses can increase if we increase the number of threads, so it should be taken in consideration if the application runs on a CPU with higher number of threads. Also we know that one way to solve this issue is run hyperthreads, so using a CPU with more number of threads per core would help with this problem.

At the end, we're achieving a 1,6 IPC per core which is 40% of the total execution capacity. The other 60% of the execution capacity is not used probably due to the slow memory accesses or slow instructions on the code.

To achieve higher performance with this application we could spend money on CPUs with more threads and higher number of hyperthreads. Also optimizing the code to achieve a performance higher than 40% IPC is a good solution, or as the version analysis and other compiler studies about CERN applications [6] have shown, recompiling the code with newer compilers might help with the IPC.

In case of trying to optimize the code, take in consideration the idea of making a SIMD version of the application. If SIMD is possible on the code, this means it has a good parallelism on a instruction level, which opens an option to a GPU version of cms-sim, since GPUs need this instruction level parallelism to be possible. Also, the memory usage is very low, which is good for GPUs, because we need to copy the memory data into GPU's data structures.

References

[1] Ttbar simulation https://twiki.cern.ch/twiki/bin/view/CMSPublic/SWGuideSimulation
[2] Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz: https://ark.intel.com/content/www/es/es/ark/products/83359/intel-xeon-processor-e5-2640-v3-20m-cache-2-60-ghz.html
[3] PrMon: https://github.com/HSF/prmon
[4] Trident: https://gitlab.cern.ch/UP/Trident
[5] Linux Perf: https://perf.wiki.kernel.org/index.php/Tutorial
[6] Caterina Marcon: https://indico.cern.ch/event/772021/