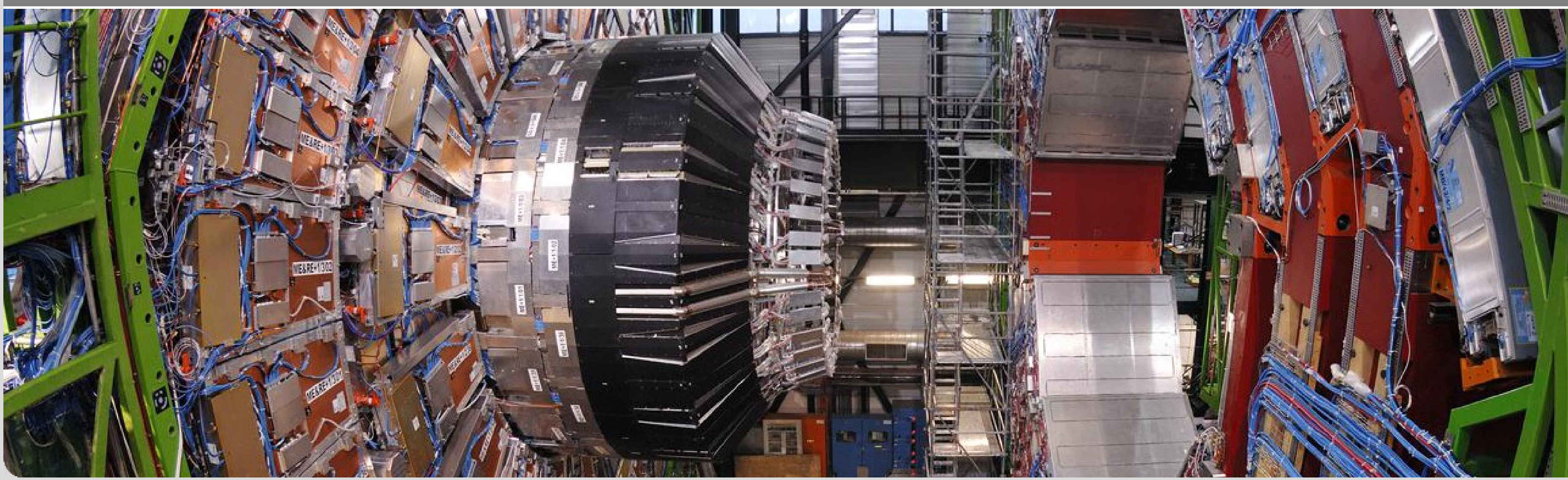
Development and Evaluation of Vectorised and Multi-Core Event Reconstruction Algorithms within the CMS Software Framework

CHEP 2012

Thomas Hauth, Danilo Piparo, Vincenzo Innocente

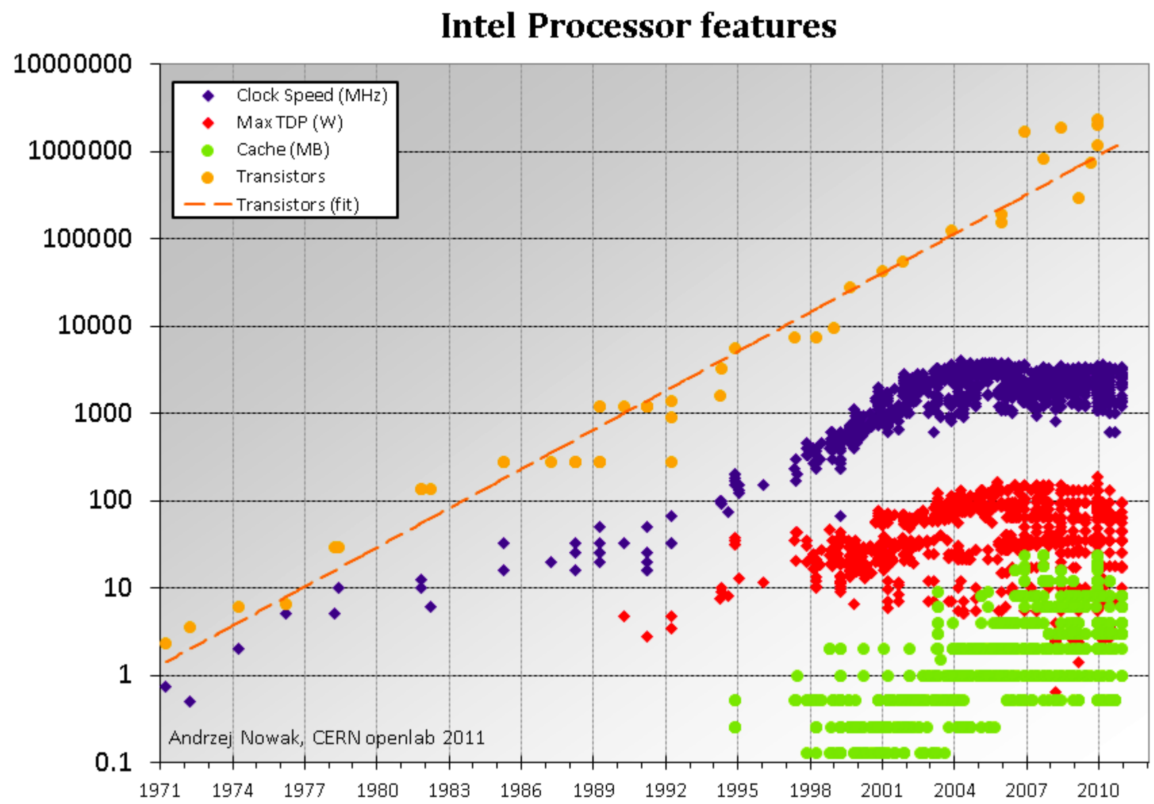


The Performance Challenge

- Since circa 6 years the single CPU clock frequency has not increased anymore:

  "The free lunch is over"

- The additional transistors are mainly used to implement:
  - More CPU Cores
  - Larger Caches
  - Larger Vector Units



Source: Andrzej Nowak – CERN OpenLab

To be able to take advantage of the available hardware, software needs to:

- Use Multi-Process / Multi-Core techniques to fully load the machine's cores
- Access the vector units provided by the machine for calculations

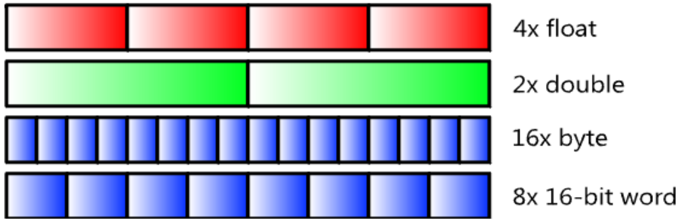
- **Vector Units in modern CPUs**
- Multi-Threading

SIMD Instructions in modern x86-64 CPUs

- Processors supporting Single Instruction, Multiple Data (SIMD) can execute ONE instruction on MULTIPLE data
- Computations are performed in dedicated parts of the processor: vector units
- Many iterations of the SIMD instruction set in x86-64 CPUs exist (MMX, SSE, SSE2, … , AVX) and newer versions feature larger register sizes

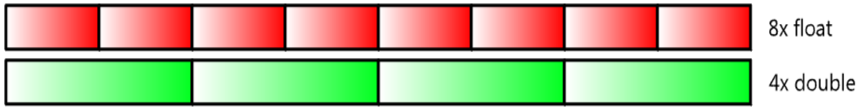
SSE2

- Virtually all CPUs since 2003
- Register Size: Two double precision floating point values



AVX

- Intel Sandy-Bridge (introduced 2011)
- Register Size: Four double precision floating point values



Intel MIC

- Register Size: Eight doubles (~2013)

SIMD is **not** multi-threading, all happens within one core !

Pictures taken from http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/

Introduction: GCC Auto-Vectorization

- Until recently, the CPU's vector units could only be utilized by interleaving the regular C++ code with explicit SIMD instructions, called intrinsics:

```
// load numbers into SIMD registers
__m256 ymm0 = _mm256_broadcast_ss(constants1);
__m256 ymm1 = _mm256_broadcast_ss(constants2);
// multiply the set of numbers and store the result in yres
__m256 yres = _mm256_mul_ps(ymm0,ymm0);
```

  This approach has some major disadvantages:

  - The code has to be re-implemented for each SIMD instruction set (SSE, AVX, ...) Including all work like debugging, verifying and profiling of the different code sections
  - Does not scale if new SIMD instructions with larger registers are introduced
  - Programming with SIMD instructions is difficult and error prone (> closer to Assembler)
  - Hard to port existing code to the SIMD instruction sets. One line in C/C++ code can easily end up in 10+ lines of SIMD instructions

Introduction: GCC Auto-Vectorization

- Recent GNU Compile Collection (GCC) versions can detect C/C++ source code fragments suitable to be processed on the vector units and can automatically compile them to SIMD instructions
- This process is called Auto-Vectorization
- However: GCC is not able to auto-vectorize most C++ source code out-of-the-box
- Some requirements must be met:
  - Predictable Loop Iterations [***no*** `while ( pVal != NULL ) { … }`]
  - No external function calls
  - Need for data structures which are continuous in memory (i.e. C-Style arrays)
  - Limited branching
- Advantages over the use of explicit SIMD instructions
  - Source code stays high-level C++
  - One version of the C++ source code for all CPU generations
  - Scales after recompiling if new SIMD instructions with larger registers are released

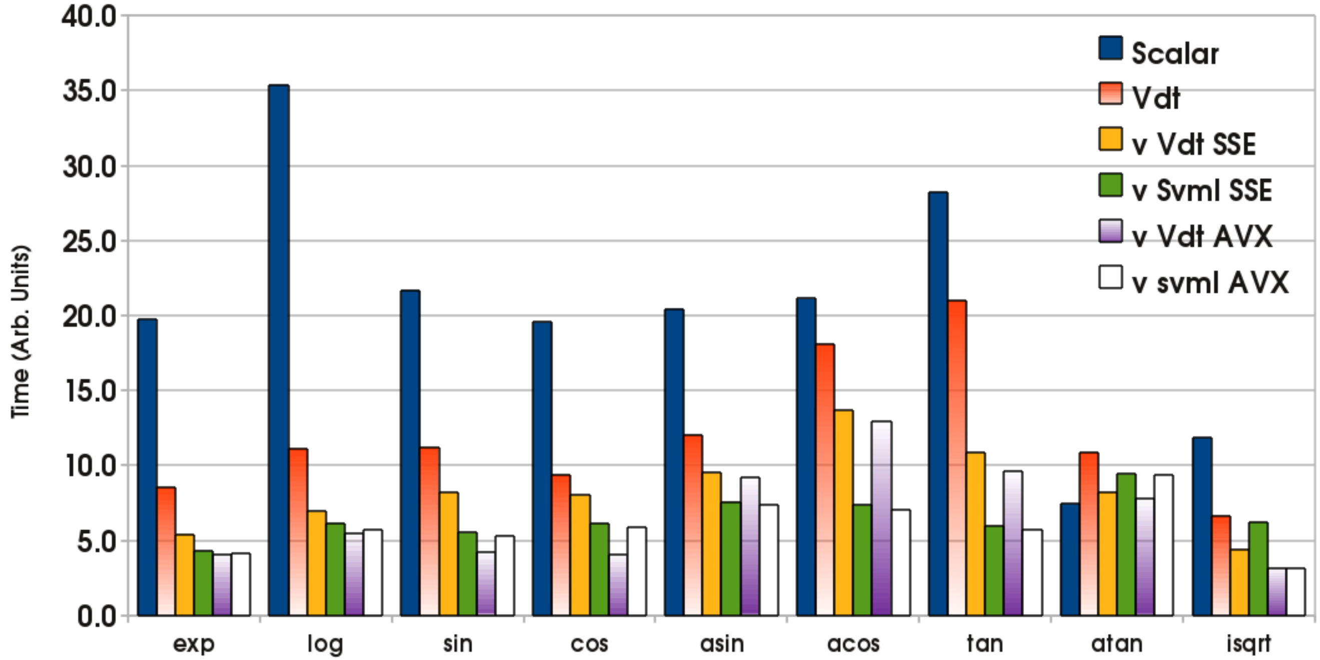
Double Precision Fast Transcendental Functions

Many open source mathematical libraries are available but...

- Only a few treat double precision numbers
- None is easily vectorizable with various SIMD instruction sets (SSE, AVX, ..)

We created a set of auto-vectorizable math functions for double precision, called **vdt math**

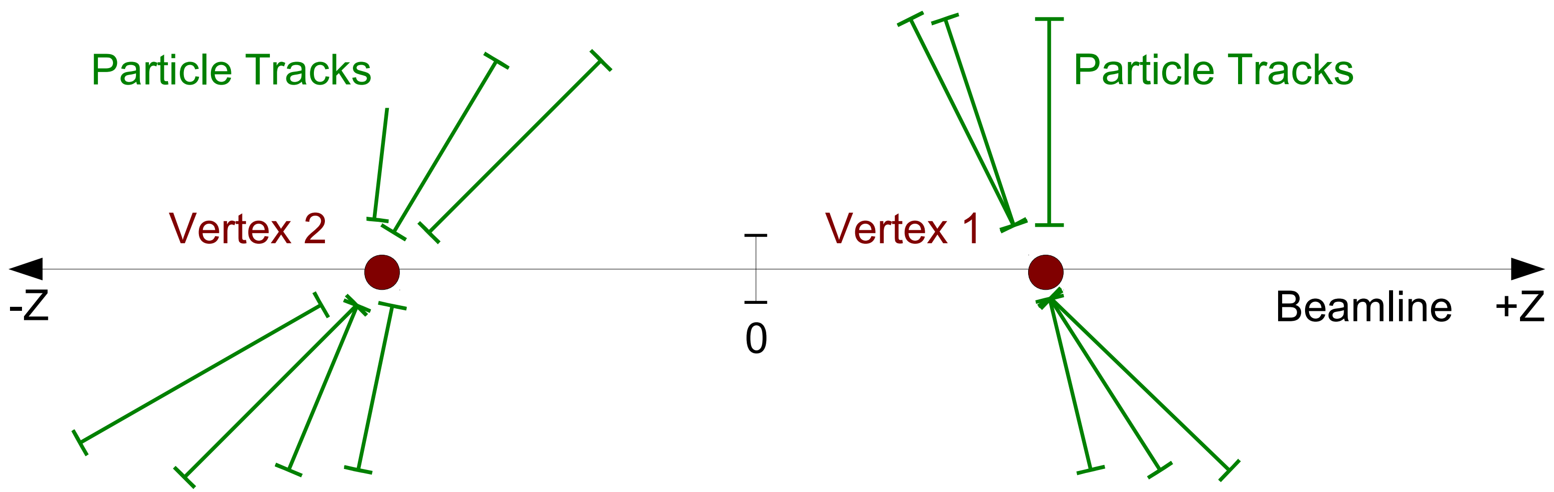
- Start from good-old Cephes library (Padé approximates)
- A multitude of useful math functions are included: `inverse square root, exp, log, sin, cos, tan, asin, acos, atan`
- Very good approximation of stdlib math functions (see backup for details)



Scalar = glibc libm

Vertex Clustering

- Part of the CMSSW Reconstruction software
- Tracks are the input and the amount and location of primary vertices along the Z-Axis is computed using the Deterministic Annealing algorithm
- Nested loops over tracks and vertices have to be performed many times → Ideal for vectorization
- This clustering step represents 3% of the overall reconstruction runtime



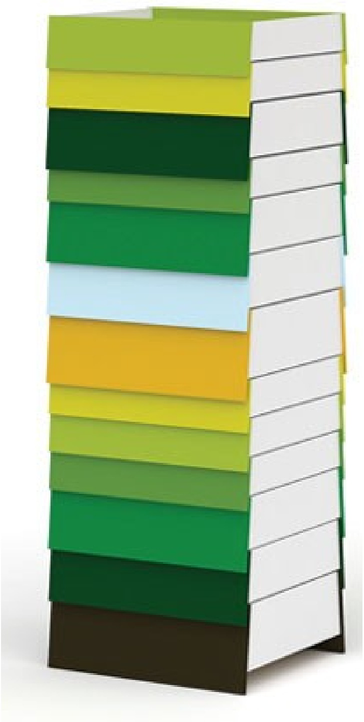
Vectorized Vertex Clustering

- Two computation intensive loops in the clustering code have been modified so they are auto-vectorized by GCC
- After vectorizing the code, 60% of the time spend in the Vertex Clustering algorithm is calculating the exponential function
- Perfect opportunity to utilize the vdt math library which provides a fast and vectorized exponential function
- By replacing the stdlib `exp()` with the vdt version and using the vectorized version of Vertex Clustering, the runtime of this module was reduced by more than a factor of two
- The physics output is identical to the regular version
- This improvement is part of the official CMSSW 5.2 release

Version	**Runtime for 50 Events [s]**	**Ratio [1]**
Regular	26.64	1.0
Vectorized	19.96	0.74
Vectorized + vdt math	11.46	0.43

Bottom Line: Vectorization

- Vector units in x86-64 CPUs are here to stay … and grow!
- Great improvements can already be achieved with the hardware we have today



Evaluation for CMS

+ Easy to use in current CMSSW setup (new compiler)
+ Can bring huge (factor 2-3) improvements for specific problems
− Hard to port some of the existing code due to the complex memory layout

We documented our results on vectorization and added educational code examples:
https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookWritingAutovectorizableCode

- Vector Units in modern CPUs
- **Multi-Threading**

Chosen Parallelization Technology: Intel TBB

- Many Threads (> Paths of execution) are run at the same time on the CPU cores
- Intel Threading Building Blocks (TBB) 4.0 update 3 Open Source (GPL license)
- Compiled with GCC 4.6.2 (default compiler of CMS Software)
- Very nice integration with C++ (in contrast to OpenMP or OpenCL):
  - Templated thread-safe containers and other data types
  - Encapsulate parallel code segments in C++11 lambda expressions
- The package provides: Loop parallelism constructs, Concurrent containers, Atomic operations and much more

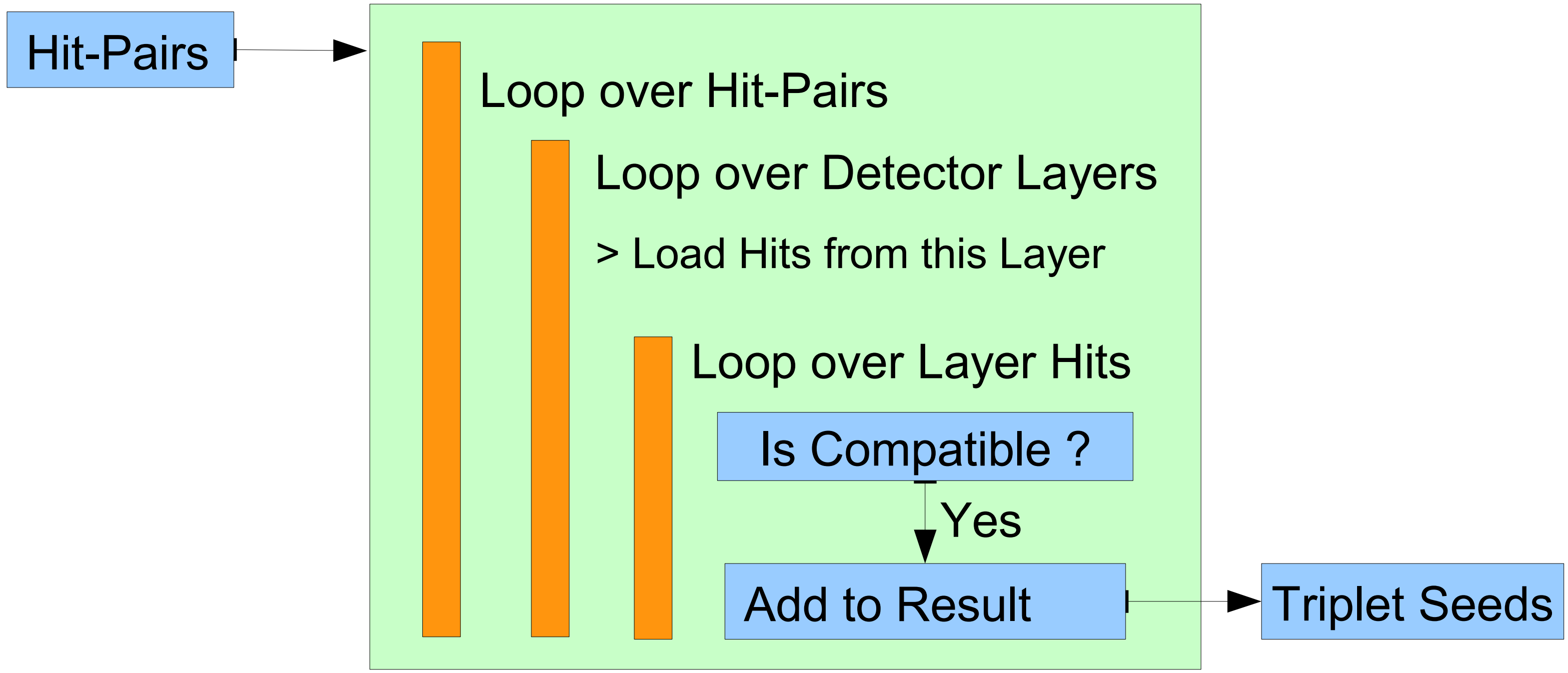
Extension of the CMS Software Framework

- A TBB Service was created which preserves a thread pool over the physics event boundaries
- The number of threads can be set in the python CMSSW configuration file
- A thread-safe reference counting was implemented using the `tbb::atomic` data type

Intel TBB website:
http://threadingbuildingblocks.org/

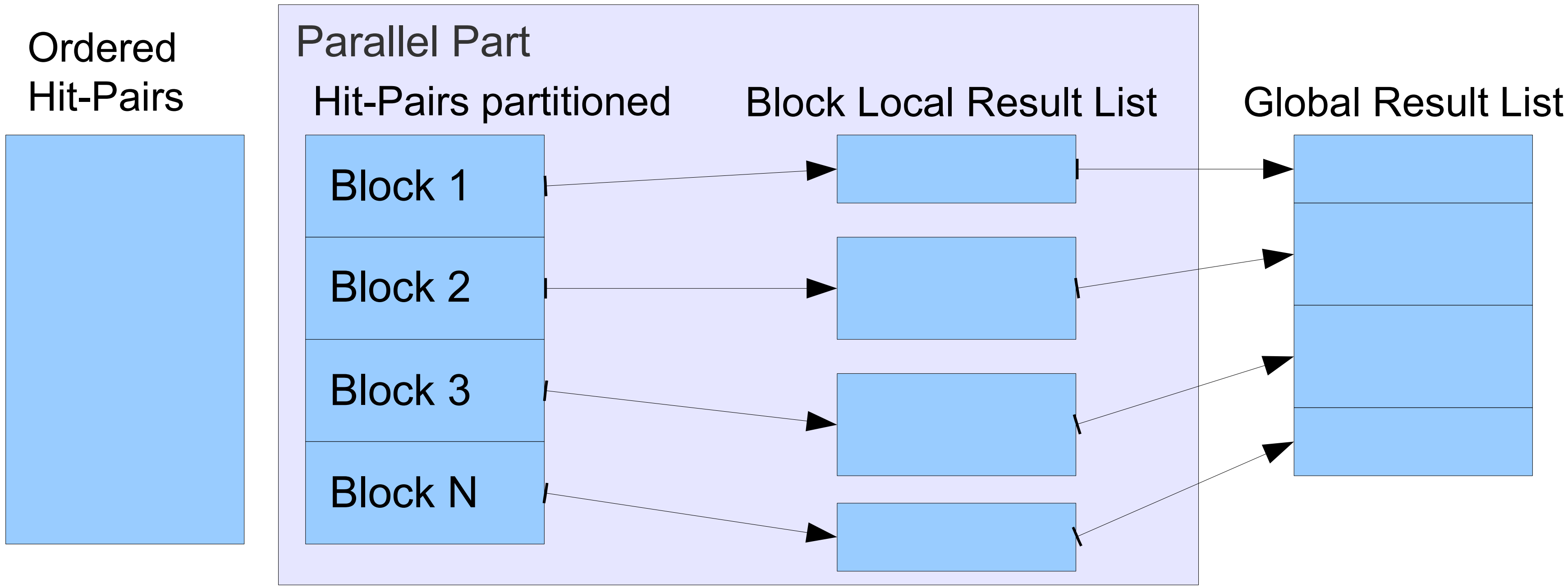
Triplet Seeding in CMS

- Energy deposits of charged particles in the CMS tracker are reconstructed as hits
- Before starting the track reconstruction, seeds from three topologically compatible hits in the tracker are searched: hit-triplets
- Starting with two hits which have been already found to be compatible (hit-pair) possible hits of subsequent tracker layers are evaluated
- This seeding procedure amounts to about 14% of the overall runtime of the CMS Reconstruction



Triplet Seeding in Parallel

- Preserving the ordering of the output collection is essential for subsequent algorithms and validation purposes
- Filling an unsorted output collection with multiple threads at the same time can result in non-reproducible results
- We used a scheme to partition the input collection of hit-pairs in equally sized blocks
- A private result list is associated with every block and is merged in the correct order into the global result list at the end of the algorithm execution. No explicit sorting needed.
- The distribution of the blocks to the available threads is handled by TBB



Validation

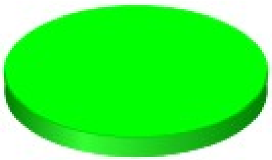
- We compared the multi-threaded version (10 threads) and the official (serial) release of CMSSW
- Considering 100 events coming from the 2011 HighPU dataset
  - Comparing bin2bin all 43k Data Quality Monitoring (DQM) histograms did not reveal any difference
  - Particle tracks parameters are 1:1 identical (momenta,chi2...)
- No crashes or segmentation faults have been observed in all test runs
- Large scale tests are of course needed but there is no reason to expect a difference

Tracking part of the complete validation procedure using DQM histograms:

Tracking

142 COMPARISONS:

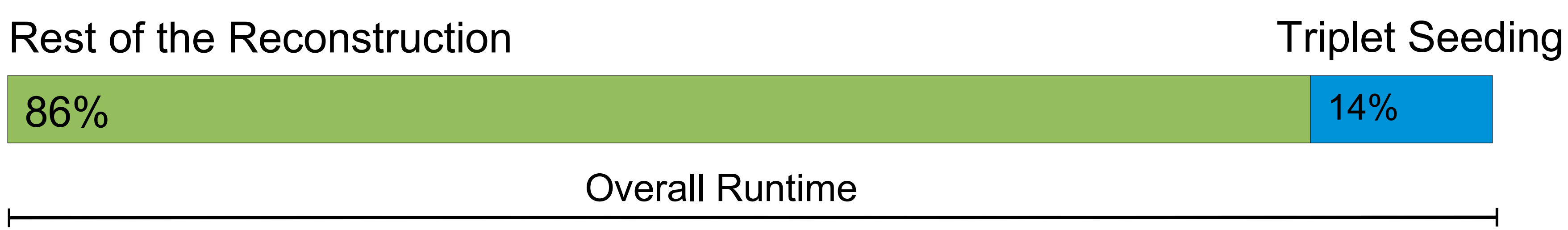
- SUCCESS: 100.0% (142)



See poster by Danilo Piparo on CMS validation this afternoon:
RelMon: A General Approach to QA, Validation and Physics Analysis through Comparison of large Sets of Histograms (ID: 211)

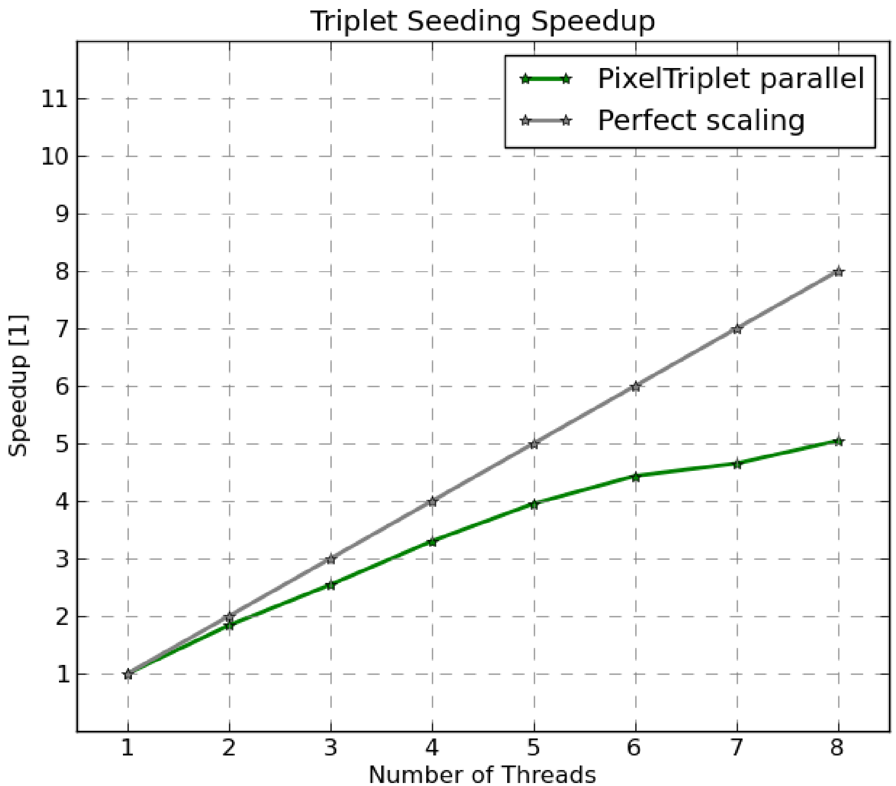
Performance Measurements

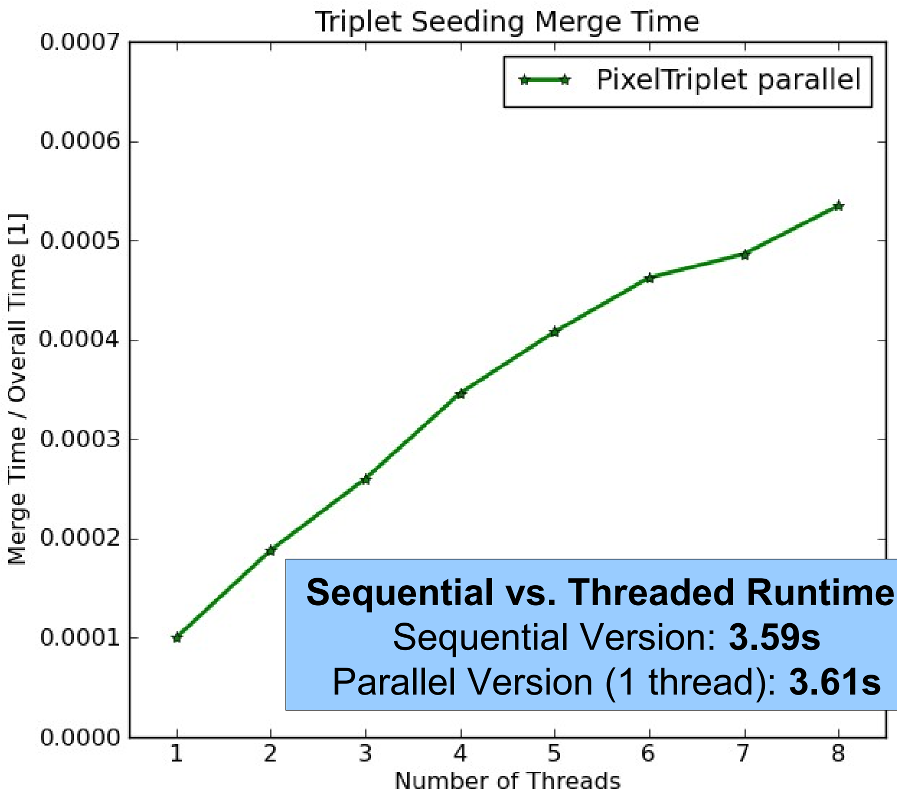
- The full CMS reconstruction chain (but: no output to disk) was run with different numbers of threads
- Input: 50 events of the highest pile-up sample recorded with the CMS detector in 2011
- On average, one event contains ~40 collisions
- Test Setup:
  - Intel(R) Core(TM) i7 CPU X 980 @ 3.33GHz with 6 physical cores (12 HyperThr.)
  - 6 GB RAM
  - Scientific Linux 5.8
  - CMSSW 5.2 official release (with modifications for the multi-threading code)
  - The measurements labeled *Serial* refer to an unchanged version of CMSSW (no TBB Service, no atomic operations)
- The triplet seeding takes about 14% of the runtime in the serial version
- Therefore, the maximum speed-up of the algorithm when running multi-threaded is 14% over the serial runtime



Triplet Seeding Runtime and Scaling

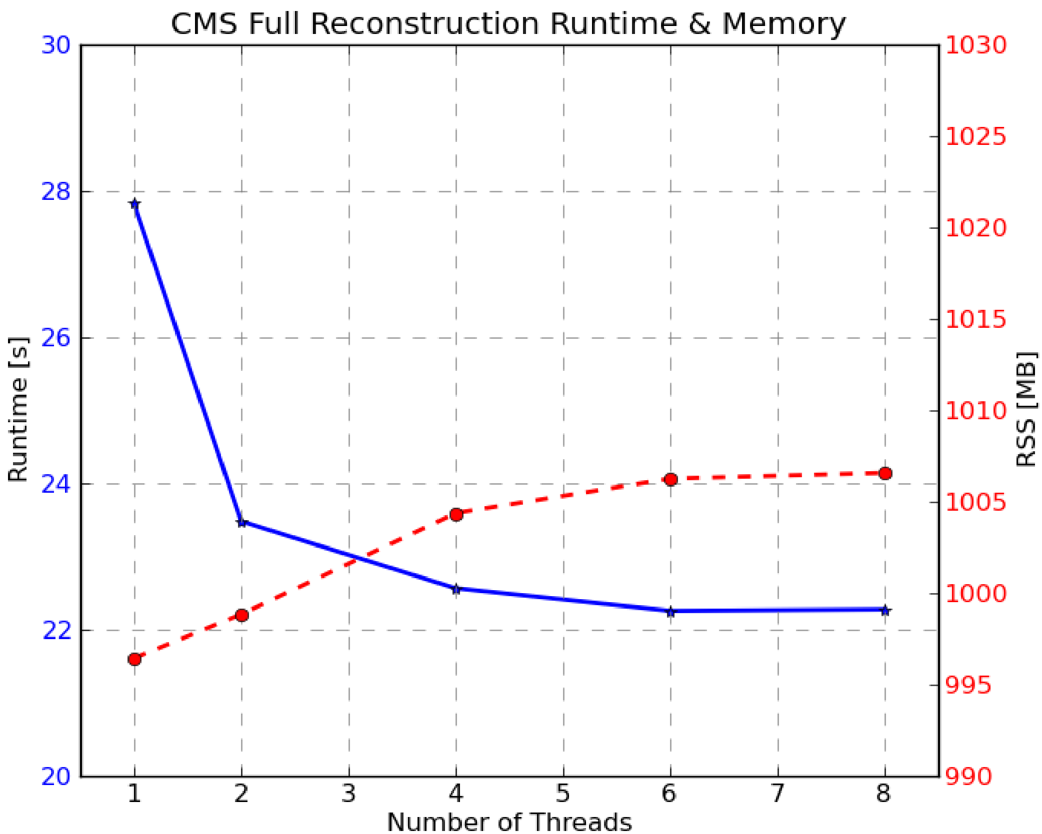
- Good scaling up to five cores
- Compared to the overall runtime of the algorithm, the final merge step only takes about .1 to .3 percent of the triplet seeding time
- This depends on the number of threads: for more threads more blocks are partitioned





Sequential vs. Threaded Runtime:
Sequential Version: **3.59s**
Parallel Version (1 thread): **3.61s**

CMS Reconstruction Runtime and Memory



- Each thread adds about 1 MB to the overall memory consumption. Negligible compared to the memory footprint of the application (~ 1 GB) > lightweight scaling
- Higher-than-expected scaling from 1 to 2 cores, probably due to the positive effects of using the L1/L2 caches of two cores simultaneously

Hyperthreading: Food for Thought

- With a multi-threaded application we can use more (Hyperthreaded) Cores with very little memory overhead

Test Scenario:
Slightly different Machine > need more RAM :)
Intel Core i7-3930K CPU at 3.20GHz
6 Physical Cores (12 Hyperthreaded)
16 GB RAM
Scientific Linux 6.2
50 High-Pileup Data Events

Runtime of **6 Single-Threaded** CMSSW Applications: **798 +/- 2 sec**

Runtime of **6 Two-Threaded** CMSSW Applications: **765 +/- 6 sec**

Using the Hyperthreading of the machine results in a decrease in runtime of 4.2 %
This number is very close the theoretical decrease of 7% with two threads. The cache benefit is not visible here, as the Hyperthreading can only use the cache of the 6 physical cores.

This is a good way to utilize the already purchased resources !

Bottom Line: Parallel Algorithms

- A multi-threaded track seeding using TBB was implemented within the CMS Software Framework
- Much more than a prototype: Tested and validated in a production environment with actual CMS proton-proton data
- Algorithm Parallelism is a feasible way to speed-up long-running modules and serial module chains

Evaluation for CMS

+ Can be applied to existing code with minor changes
+ Prepares our software for next-generation accelerators (Intel MIC)
+ Wide varieties of processing can be run in parallel (Tracks, Hits, ...)
− Ensuring concurrent data-access in the current framework is essential. Efforts are underway to simplify this for the algorithm developer.

Summary

- First production-ready implementations of CMS Algorithms using parallelization presented
- If applied with care and savvy, the physics quality is preserved but large speedups can be achieved
- **Auto-vectorization** is best suited for
  - Computations on data structures continuous in memory (i.e. C-Style arrays)
  - Works best if all used code is contained in one method, inlining code can help
- **Thread-Parallelization** is best suited for
  - Compute intensive loops with sufficient amount of input data
  - Works across method boundaries, as long as the used data structures are accessed in a thread-safe manner
  - Can also be applied on a Framework level
    - see talk by Christopher Jones (CMS) on Monday: Study of a Fine Grained Threaded Framework Design, Contribution ID: 194

To preserve the excellent performance of CMS in the future (Detector Upgrade, SuperLHC) we have to take into consideration these new software technologies

BACKUP

Toy Example: Evaluation of a Polynomial

- 3rd-Order Polynomial is calculated for an array of input values

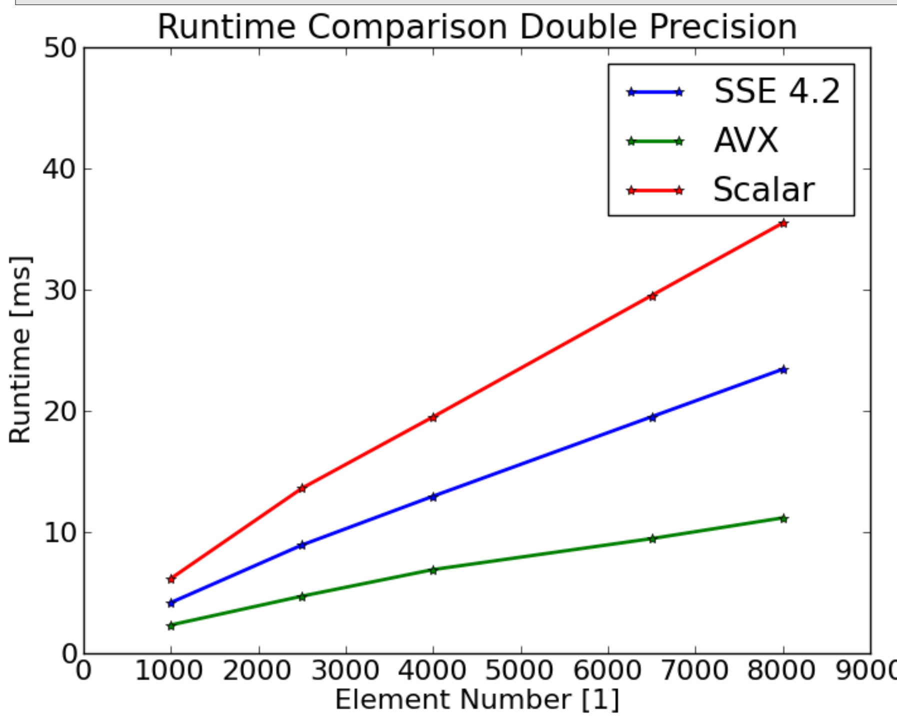
$$y(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

- The same C++ code is compiled with GCC 4.6.2 in three flavours:
  - Scalar version (no vector units)
  - SSE 4.2 (2 doubles in parallel)
  - AVX (4 doubles in parallel)

- The plot on the right shows the overall runtime for various sizes of the input array for double precision values
- The gain in performance for the SSE and AVX versions are clearly visible and almost approaches the theoretical limit.

C++ Code Excerpt

```
...
for ( size_t i = 0; i < ArraySize; ++i)
{
  y[i] = a_3 * ( x[i] * x[i] * x[i] )
       + a_2 * ( x[i] * x[i])
       + a_1 *   x[i] + a_0;
}
...
```



VDT Accuracy

Table 2. The interval of defintion and accuracy of `Vdt` with respect to the corresponding `libm` implementation. The accuracy was estimated evaluating the functions over one million randomly distributed numbers in the domain. The superior and the average of the most significant different bit are reported. A difference of zero means identity of the two numbers, considering all the bits of their representations.

Function	Interval of Definition	Superior	Average
exp	[-708,708]	2	0.14
log	(0,1e307]	2	0.37
sin	[-2π,2π]	21	1.2
cos	[-2π,2π]	21	1.3
asin	[-1,1]	2	0.32
acos	[-1,1]	8	0.45
tan	[0,2π]	21	2.1
atan	[-1e307,1e307]	0	0
inv. sqrt.	(0,1e307]	2	0.48

Framework and Algorithm Parallelism
Beyond Event Level Parallelism

- Framework Parallelism
  - After modifications (declaring dependencies etc.), parallel execution of already existing serial modules is possible
  - Hides most of the multi-threading complexity from the module developer
  - Scales very well at the price of loading and writing multiple events at the same time. See the presentation by Chris Jones*
- Algorithm Parallelism
  - Changes mostly contained in one module
  - Very lightweight scaling (in terms of memory)
  - Transparent to subsequent Modules
  - Most profitable to apply on long-running Modules which can only operate sequentially (like CMS Iterative Tracking)

A great potential lies in combining these two levels of parallelism: scale with the amount of input data and the number of available computing cores.

* Forum on Concurrent Programming Models and Frameworks, 14.03.2012
http://indico.cern.ch/conferenceDisplay.py?confId=181721

How to ensure thread-safe code ?

- High quality of CMSSW code base helps, **`const`**-correctness enforced everywhere
- **`const`** is your friend:
  - **`const`** objects and methods can be accessed safely
  - But not always: C++ **`mutable`** keyword
  - Non-const variables can be assigned to a **`const`** reference to ensure safe access within the mutli-threaded code section:

```
AClass aobject(size);
AClass const& aobject_threadsafe = aobject;
```

- Use of TBB concurrent containers whenever multi-threaded write access to collections is necessary
- **`tbb::atomic`** data type was used to ensure thread safe reference counting
- Ultima-Ratio: Explicit Locking
- Software Tools for big applications:
  - Helgrind (part of valgrind) was tested on a simple example outside of CMSSW, but produced **many** false positives
  - Suggestions or hints are very welcome
- Use the serial implementation and run a lot of multi-threaded validation, check for crashes and compare the outputs