WLCG
Worldwide LHC Computing Grid

HEPiX

CPU benchmarking and SIMD vectorization

A few preliminary ideas

Andrea Valassi (IT-SC-RD)

HEP-SCORE Deployment Task Force meeting, 6th June 2021
https://indico.cern.ch/event/1030673/

Disclaimer

- Not an expert: only started coding vectorized code less than one year ago
  - Madgraph event generator: https://doi.org/10.5281/zenodo.4785174

- Not much time to prepare this talk or discuss it with others yet – sorry

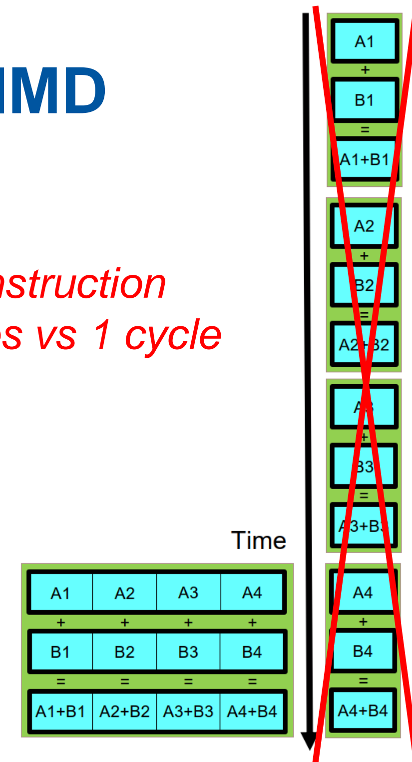
- No detailed discussion yet on vectorization in the benchmarking WG
  - *...only few HEP software workloads exploit CPU vectorization today...*

- Just a few points for discussion, essentially...
  - NB: these are my personal opinions/suggestions, not those of LHCb

Many thanks to Sebastien Ponce, Marco Clemencic, Hadrien Grasland for their help with SIMD development

Vectorization – the hardware view: SIMD

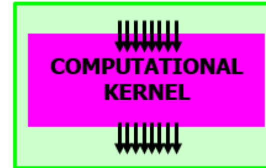
- SIMD: Single Instruction Multiple Data
  - The ability of a processor to *execute several operations in a single instruction*
  - e.g. A1+B1=C1, ... A4+B4=C4 vs. A[1...4]+B[1...4]=C[1...4] – *4 cycles vs 1 cycle*
- Simplifying: two main ingredients
  - Vector registers storing multiple data
  - Instruction set supporting vector operations on those registers
- There are several sets of advanced vector extensions, e.g.
  - SSE4.2: 128-bit wide xmm registers (2 doubles, 4 floats)
  - AVX2: 256-bit wide ymm registers (4 doubles, 8 floats)
  - AVX512: 512-bit wide zmm registers (8 doubles, 16 floats)
  - In theory, a speedup (or throughput increase) by up to 16x is possible...
- Different processors support different levels of SIMD (check /proc/cpuinfo)
  - *Should we benchmark this? Should a system have a higher score because of its SIMD?*



NB: this is orthogonal to the multi-core speedup (multi-processing, multi-threading)

Vectorization – the software view: data parallelism (lockstep)

- Exploiting SIMD requires computational workflows with excellent data parallelism
  - *Compute the same function on multiple data at the same time (in lockstep)*
    - And there must be little serial code outside this kernel (Amdahl's law...)
  - A similar challenge exists for GPU software programming



- *Coding to exploit SIMD on CPUs is hard* (harder than coding for GPUs)
  - The wrong memory layout may be a blocker (Array-of-Structure vs Structure-of-Array)

- As a consequence, *CPU SIMD is heavily under-exploited in HEP workloads today*
  - SIM (detector simulation): very hard to exploit because of stochastic branching
  - RECO (event reconstruction): a lot of WIP (e.g. in LHCb), not yet highly visible effects
  - GEN (event generation): just starting, good potential (IMO)
  - Analysis: good exploitation in some packages (IIUC) – but not in HEP-workloads
  - In my understanding, there is little vectorization in our HEP-score candidates

NB: achieving a multi-core speedup is much easier
  - In the worst case (and if you have enough RAM), you just run one ST application per core
  - This is exactly what we do to benchmark systems in HEP-workloads and HEP-score

CPU throughput results (2)
Double, C++ – Scalar vs SIMD

- *SIMD: excellent speedup from vectorization*
  - NB: only measuring the parallel calculation
  - Lower overall speedup (Amdahl's law...)
- Best throughput: AVX512 limited to 256-bit width
  - *x3.7 over scalar C++ (vs x4 theoretical maximum)*
    - *Estimate a x3.3 speedup over scalar Fortran*
  - Thanks to Sebastien Ponce for the suggestion!
- Disappointing: AVX512 with 512-bit width
  - Slower than AVX2, why? Slower clock, what else?
  - Can be improved? x8 theoretical maximum...

Implementation ($e^+e^- \to \mu^+\mu^-$)	MEs / second Double
1-core MadEvent Fortran scalar	1.50E6 (x1.15)
1-core Standalone C++ scalar	1.31E6 **(x1.00)**
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles)	2.52E6 **(x1.9)**
1-core Standalone C++ 256-bit AVX2 (x4 doubles)	4.58E6 **(x3.5)**
1-core Standalone C++ "256-bit" AVX512 (x4 doubles)	4.91E6 **(x3.7)**
1-core Standalone C++ 512-bit AVX512 (x8 doubles)	3.74E6 **(x2.9)**

# Symbols in .o / Build type	SSE4.2 (xmm)	AVX2 (ymm)	AVX512 (ymm)	AVX512 (zmm)
Scalar	614	0	0	0
SSE4.2	3274	0	0	0
AVX2	0	2746	0	0
256-bit AVX512	0	2572	95	0
512-bit AVX512	0	1127	205	2045

A few AVX512VL symbols yield a 7% improvement over pure AVX2

Degree of vectorization checked by disassembling (objdump)
Custom categorization of symbols

CPU throughput results (3)
C++, SIMD – Double vs Float

Implementation ($e^+e^- \to \mu^+\mu^-$)	MEs / second Double	MEs / second Float
1-core MadEvent Fortran scalar	1.50E6 (x1.15)	---
1-core Standalone C++ scalar	1.31E6 **(x1.00)**	1.21E6 **(x0.92)** ***[x1.00]***
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles, x4 floats)	2.52E6 (x1.9)	4.50E6 **(x3.4)** ***[x3.7]***
1-core Standalone C++ 256-bit AVX2 (x4 doubles, x8 floats)	4.58E6 (x3.5)	8.17E6 **(x6.2)** ***[x6.8]***
1-core Standalone C++ "256-bit" AVX512 (x4 doubles, x8 floats)	4.91E6 (x3.7)	8.84E6 **(x6.7)** ***[x7.3]***
1-core Standalone C++ 512-bit AVX512 (x8 doubles, x16 floats)	3.74E6 (x2.9)	7.42E6 **(x5.7)** ***[x6.1]***

- Scalar: float slower than double
  - To be understood (8% effect)
- *SIMD: float ~ x2 better than double!*
  - Execute ½ as many vector instructions
  - Best throughput: 256-bit AVX512 *(x7.3 speedup against x8 theoretical maximum)*
- *Is single precision enough for physics?* Can we improve numerical stability?
  - Observed a few NaN every million MEs when using single precision
  - Using fast math (~x2 speedup) also requires excellent control of numerical stability

Vectorization – compilation flags and builds

- Most of the "usual" compiler flags affect all systems in "similar" ways
  - Example: -O3 is an optimization producing code valid for all platforms
  - (Counter-)Example: -m64 produces code valid for all 64bit platforms (ALL modern ones!)

- *Compiler flags for vectorization are special – they are architecture-specific*
  - Example: -march=haswell (SSE4), -march=nehalem (AVX2), -march=skylake-avx512
  - If you run code compiled for AVX512 on a CPU that does not support it, it will crash

- *The experiments generally have separate builds for different SIMD levels?*
  - This is certainly true at least in LHCb: "x86_64+avx2+fma-centos7-gcc9-opt"
    - Eventually move to microarchitecture feature levels (e.g. x86-64, x86-64-v1 etc...)?
  - *To my knowledge, none of the current HEP-score candidate workloads uses a SIMD build*

- Alternative: "fat" binaries, supporting multiple SIMD levels with an internal trampoline
  - It is easy to determine from the code itself which SIMD features a host system supports

- Two options for future containers of vectorized HEP software workloads?
  - Embed several builds in one container, choose "best" SIMD level in the calling bash script
  - Embed a single "fat" executable, let it choose the "best" SIMD level
  - In any case: measure and record scores for several SIMD levels (not only the "best" one)

Benchmarking – a problem with many degrees of freedom (1)

- A gedanken experiment – in theory, very peculiar situations are possible
  - Suppose system A supports no SIMD, while system B supports AVX2
  - Suppose workload W1 uses no SIMD, while workloads W2 and W3 can be highly vectorized
    - Scores from W1 are a1 and b1, with b1/a1 = c *(i.e. B/A = c)*
    - Scores from W2 (W3) in scalar mode are a2 (a3) and b2 (b3), with b2/a2 ~ b3/a3 ~ c *(i.e. B/A ~ c)*
    - Scores from W2 with vectorization are a2 and 4x b2 *(i.e. B/A ~ 4c!)* – AVX2 fits 4 doubles
    - Scores from W3 with vectorization are a3 and 8x b3 *(i.e. B/A ~ 8c!)* – AVX2 fits 8 floats
  - Workloads W2 and W3 dramatically prefer system B, as only system B provides SIMD...
  - *Should we benchmark B/A as c? or as 4c? or as 8c?...*

- The situation is currently very unlikely (... but things may change?...)
  - There is not a large spread of SIMD capabilities in WLCG (most are AVX2? some SSE4?)
  - We do not have HEP software workloads with such high benefits from SIMD
    - And even if we had, their relative weight in the HEP-score average would be small

Nevertheless: "Both the press and the customer must be informed about the danger and the folly of relying on either a single performance number or a single benchmark"

[Kaivalya Dixit, former president of the SPEC corporation]

A single number (HEP-Score) is what we want, and for good reasons – but we should be careful!

Benchmarking – a problem with many degrees of freedom (2)

A glimpse at GPUs

- Not related to SIMD vectorization – but still related to float vs double
- Floating-point operations handled by different units for float (FP32) and double (FP64)
  - Typically: ratio of FP32 to FP64 units is 2 for data center products, 32 for consumer cards!
  - *This is why GPU specs typically report TWO numbers: float Flops and double Flops...*

GPU throughput results (2)

CUDA – Double vs Float (and NVidia V100 vs T4)

- *V100: float ~ x2.2 better than double!*
  - Similar to CPU SIMD, different reasons
  - V100 Flops (&cores): FP32 = *2x* FP64
  - Fewer registers: float=48, double=120
- *T4: very limited double performance*
  - T4 Flops: FP32 = *32x* FP64
  - May be even worse in consumer cards

Implementation ($e^+e^- \to \mu^+\mu^-$)	MEs / second Double	MEs / second Float
1-core MadEvent Fortran scalar	1.50E6 (x1.15)	---
1-core Standalone C++ scalar	1.31E6 **(x1.00)**	1.21E6 (x0.92)
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles, x4 floats)	2.52E6 (x1.9)	4.50E6 (x3.4)
1-core Standalone C++ 256-bit AVX2 (x4 doubles, x8 floats)	4.58E6 (x3.5)	8.17E6 (x6.2)
1-core Standalone C++ "256-bit" AVX512 (x4 doubles, x8 floats)	4.91E6 (x3.7)	8.84E6 (x6.7)
1-core Standalone C++ 512-bit AVX512 (x8 doubles, x16 floats)	3.74E6 (x2.9)	7.42E6 (x5.7)
Standalone CUDA NVidia V100S-PCIE-32GB (TFlops*: 7.1 FP64, 14.1 FP32)	7.25E8 **(x550)**	1.59E9 **(x1200)**
Standalone CUDA NVidia T4 (TFlops*: 0.25 FP64, 8.1 FP32)	3.21E7 **(x25)**	6.52E8 **(x500)**

* https://www.techpowerup.com/gpu-specs/tesla-t4.c3316
https://www.techpowerup.com/gpu-specs/tesla-v100-pcie-32-gb.c3184

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs vCHEP – 19 May 2021 20

Conclusions

- In theory, vectorization could dramatically alter a system CPU benchmark
  – Throughput ratios of two systems may vary by factors for vectorized software workloads
- *In practice, this is not a problem for HEP software workloads on WLCG today?*
  – HEP software is largely not vectorized, WLCG hardware is relatively uniform?
- Reminder: choosing a single-number benchmark is hard (will be worse for GPUs!)
  – Multi-dimensional problem: vectorization is a typical example, but not the only one...
- Suggestions for the current HEP-score candidate?
  – Use the software workloads as the experiments have already packaged them
  – LHCb (GEN-SIM) is certainly not vectorized
  – I assume the others are also not vectorized? (or internally choose/assume a SIMD level?)
- Suggestions for future HEP-score versions? (When there are vectorized workloads!)
  – Allow each container to run different SIMD builds (different executables or a “fat” binary)
  – Measure/store different vectorization scores for each workload for different SIMD levels
  – On a specific system, let each workload choose the SIMD level it prefers
  – Aside – we should clarify the SIMD capabilities of the hardware we already have in WLCG?

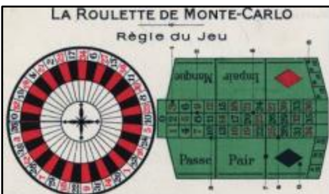
Backup slides

Main design idea: event-level data parallelism (lockstep)

- In MC generators, all events *in one channel* initially go through the same calculations
  - *Computing MEs involves the calculation of the exact same function on different data points*
  - This is what *makes event generators a good fit for GPUs (SIMT) and vector CPUs (SIMD)*

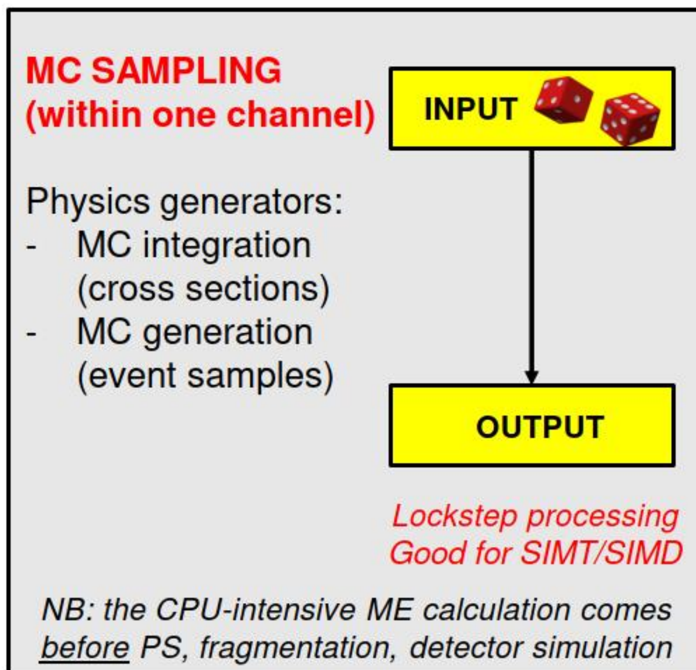
GPU SIMT (Single Instruction Multiple Threads)
Lockstep: all threads in a warp follow the same branch
Minimum parallelism: 32 threads in a warp (NVidia)

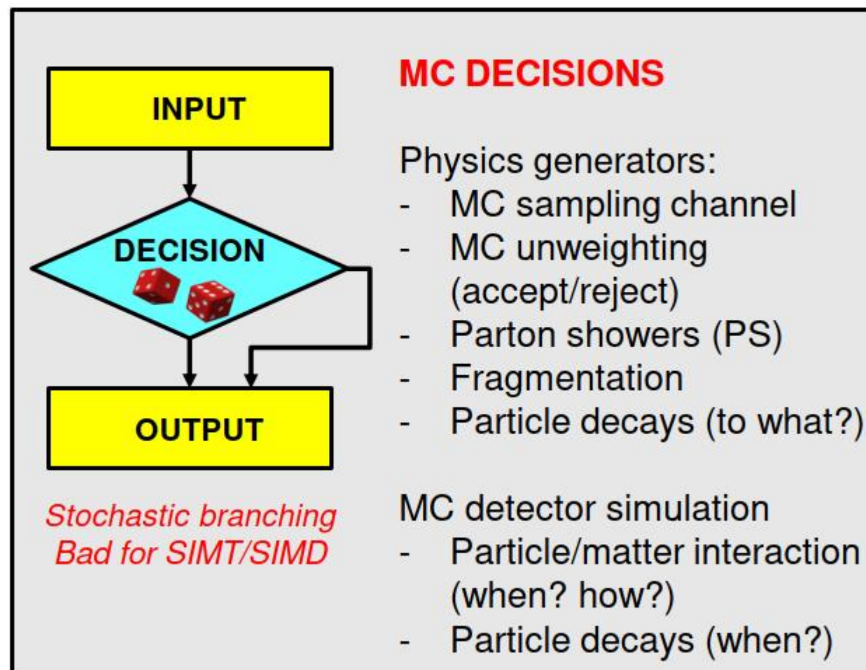
CPU SIMD (Single Instruction Multiple Data)
Lockstep: same op for all data in a vector register
Minimum parallelism: 2 to 16 (SSE/AVX2/AVX512...)



Aside – Monte Carlo's: what about branching?

- *Monte Carlo methods are based on drawing (pseudo-)random numbers*: a dice throw
- From a software workflow point of view, these are used in *two rather different cases*:





A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   11