Challenges and Opportunities given new Hardware

Stephan Hageboeck, CERN IT

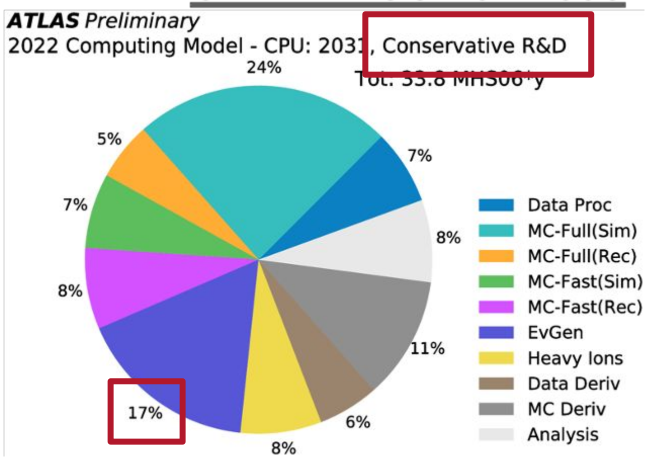
Nov 2023

stephan.hageboeck@cern.ch

Motivation: MC Event Generators and LHC

- HL-LHC computing needs expected to outgrow resource growth
  - Need R&D on software to improve efficiency and port it to new resources, such as GPUs at HPC centres
- MC generators rarely adapted to vector computations or GPUs
- Potential for more events given the same hardware

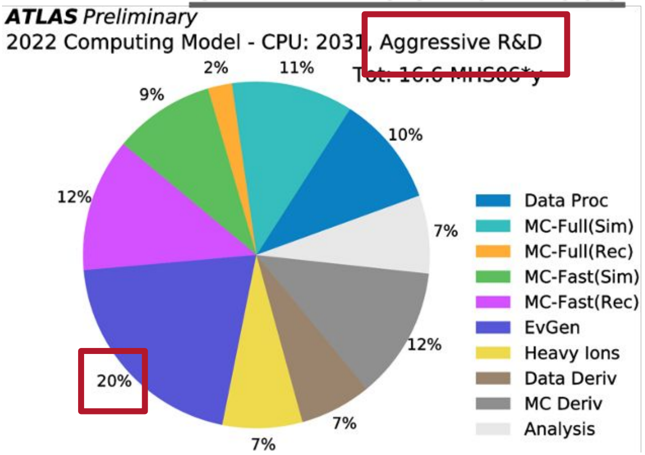
CERN-LHCC-2022-005



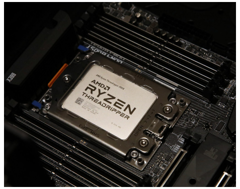
Motivation: MC Event Generators and LHC

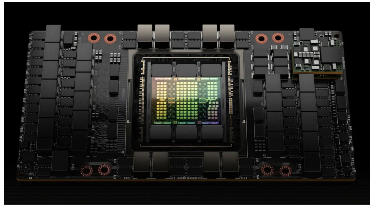
- ▶ HL-LHC computing needs expected to outgrow resource growth
    - Need R&D on software to improve efficiency and port it to new resources, such as GPUs at HPC centres
- ▶ MC generators rarely adapted to vector computations or GPUs
- ▶ Potential for more events given the same hardware

CERN-LHCC-2022-005



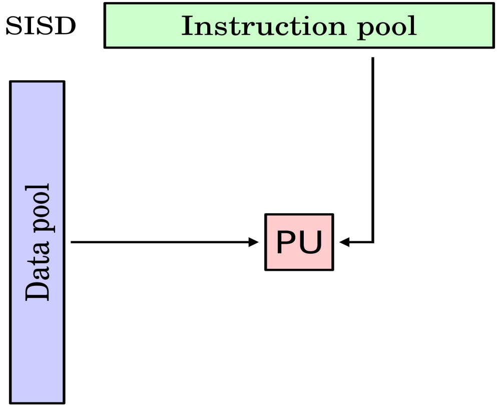
Some Hardware Examples





	AMD Ryzen Threadripper 3990X	Nvidia Tesla T4	Nvidia Tesla H100 PCIe
Cores	64	2560 (**40 SMs**)	7296 (**114 SMs**)
Max Clock Rate	4300 MHz	1590 MHz	1755 MHz
Theoretical Single (Double) Prec. Perf.	13 TFLOPS (1/2)	8.1 TFLOPS (1/32)	48 TFLOPS (1/2)
Memory Bandwidth	95 GB/s	300 GB/s	2000 GB/s
TDP	280 W	70 W	350 W

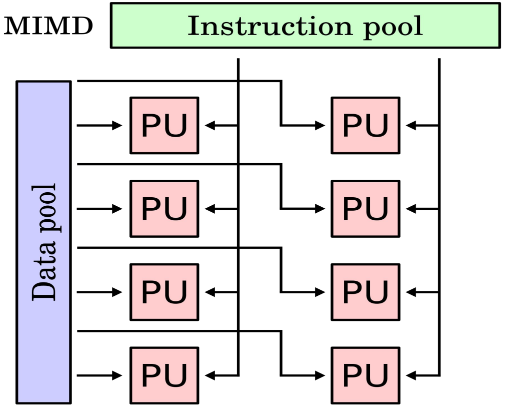
SIMD and GPUs in Flynn's Taxonomy



Single Instruction Single Data

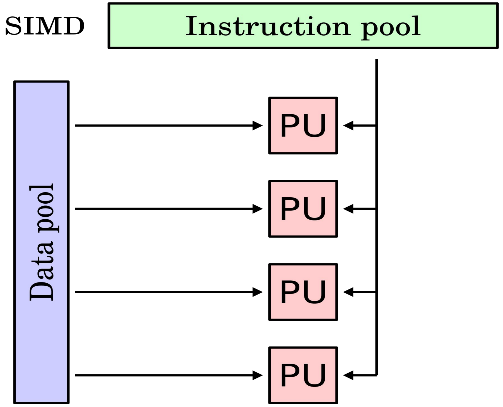
- Single core executing one instruction after the other

en.wikipedia.org/wiki/Flynn's_taxonomy



Multiple Instruction Multiple Data

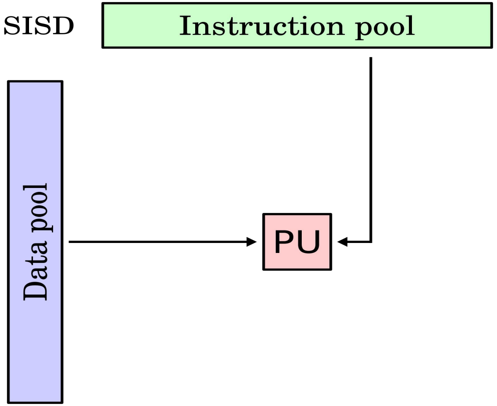
- E.g. multi-core processor
- Multiple independent threads of execution



Single Instruction Multiple Data / SIMT: "Multiple Threads"

- **GPUs** / vector units on CPUs
- Many threads running the same instruction

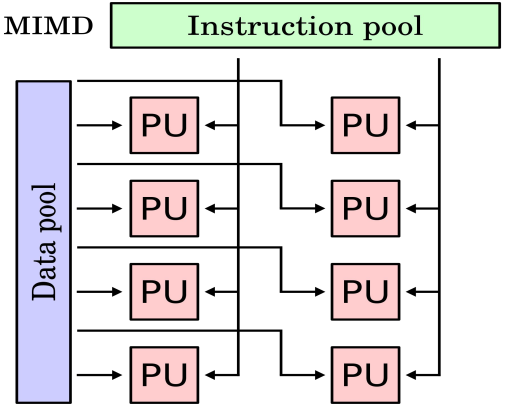
SIMD and GPUs in Flynn's Taxonomy



Single Instruction Single Data

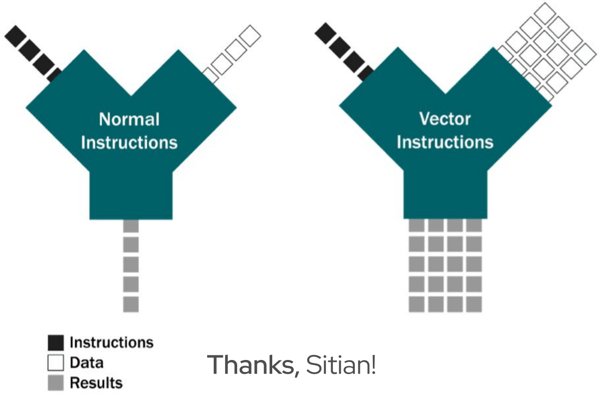
- Single core executing one instruction after the other

en.wikipedia.org/wiki/Flynn's_taxonomy



Multiple Instruction Multiple Data

- E.g. multi-core processor
- Multiple independent threads of execution



Thanks, Sitian!

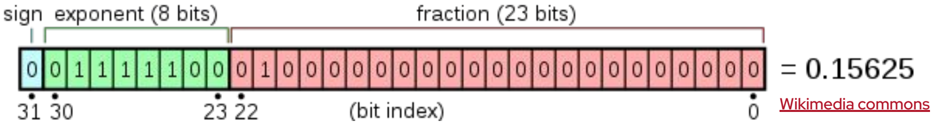
Single Instruction Multiple Data / SIMT: "Multiple Threads"

- **GPUs** / vector units on **CPUs**
- Many threads running the same instruction

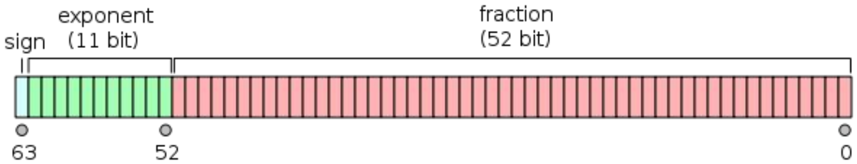
Interlude: Single vs Double Precision

- Single precision: 32 bit
- Exactly represents integers between 0 and 16,777,216 (1.68E7)
- Max: 3.4E38
- GPU "with X TFLOPS" usually means single precision

- Double precision: 64 bit
- Exactly represents integers up to 9,007,199,254,740,992 (9E15)
- Max: 1.8E308
- With "X TFLOPs single precision", you might get in double:
  - **1/32 for consumer grade**
  - **½ for data-centre GPUs**

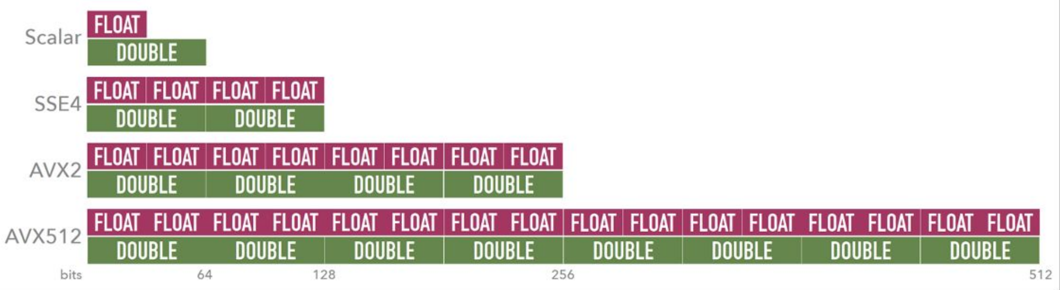


[Wikimedia commons](https://commons.wikimedia.org)



Where can we gain?

- Scenario 1: An event generator runs on a "normal" CPU
  - If the generator supports AVX2 vectorisation, we can get 4x
  - If the CPU supports certain AVX512 versions, 8x is possible (reality is complicated, though)
  - If the generator can run in single precision, double the above

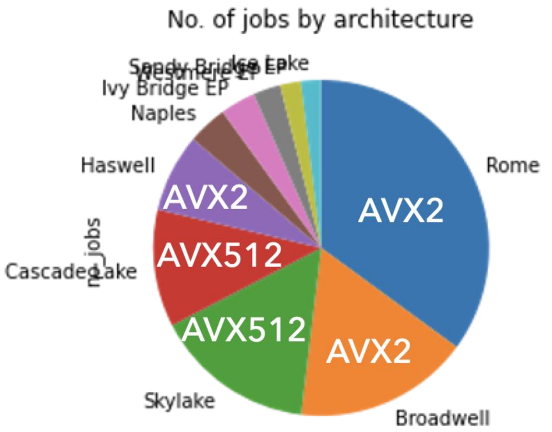

Scalar
FLOAT
DOUBLE
SSE4
FLOAT FLOAT FLOAT FLOAT
DOUBLE DOUBLE
AVX2
FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT
DOUBLE DOUBLE DOUBLE DOUBLE
AVX512
FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT FLOAT
DOUBLE DOUBLE DOUBLE DOUBLE DOUBLE DOUBLE DOUBLE DOUBLE
bits
64
128
256
512

- Scenario 2: The event generator runs on a machine with a GPU
  - Achievable speedup heavily depends on CPU, GPU and workflow running
  - Anything between no speedup and ~ 100x is possible (→ double-precision perf. of GPU?)
  - HPCs provide most compute power (FLOPS) with GPUs
  - If the event generator is not GPU-ready, a large fraction of the hardware remains unused

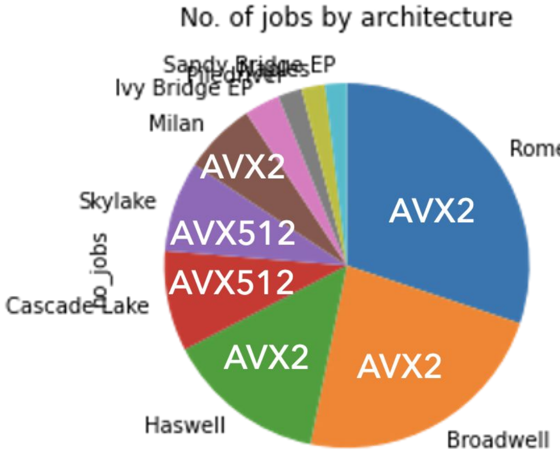
WLCG: We can have 4x speed up today

- The Grid supports AVX2 (almost) universally
- First AVX2 CPUs are from 2013
- Opportunity for 4x speedup unused if we don't adapt for SIMD
- In single precision, we could have 8x speedup

ATLAS



CMS

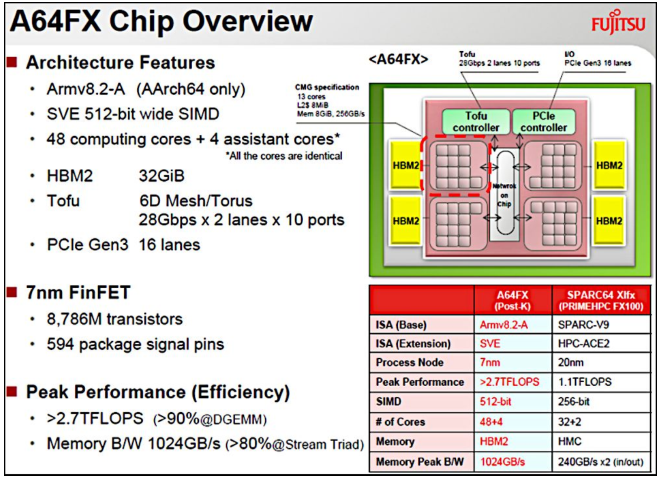


A. Sciaba, 2022
https://indico.cern.ch/event/1072141/

And in the future … ?

- AMD GPUs power Frontier, the #1 Top500
- Computing on ARM with vectorisation?
  - Fugaku #2 runs A64FX with 512 bit vectors
- LUMI #3 is running AMD GPUs
- Grace Hopper CPU/GPU hybrid
  - Arm Neoverse CPUs with 128 bit vector registers
  - Hopper GPU
- Computing hardware will become more diverse, so we want event generators that are parallel and vectorised

https://hacarus.com/ai-lab/20210405-fugaku/



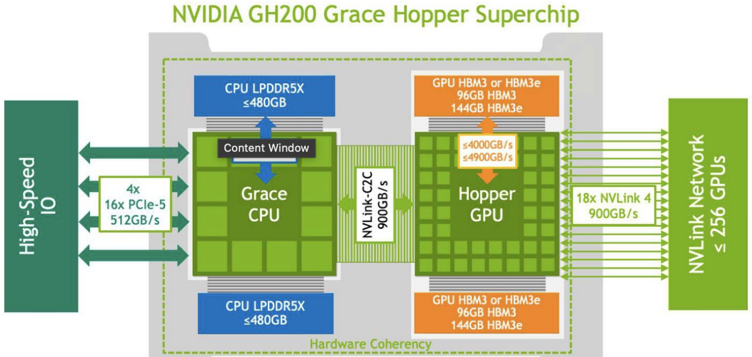
A64FX Chip Overview

FUJITSU

- Architecture Features
  - Armv8.2-A (AArch64 only)
  - SVE 512-bit wide SIMD
  - 48 computing cores + 4 assistant cores*

    *All the cores are identical
  - HBM2 32GiB
  - Tofu 6D Mesh/Torus 28Gbps x 2 lanes x 10 ports
  - PCIe Gen3 16 lanes
- 7nm FinFET
  - 8,786M transistors
  - 594 package signal pins
- Peak Performance (Efficiency)
  - >2.7TFLOPS (>90%@DGEMM)
  - Memory B/W 1024GB/s (>80%@Stream Triad)

	A64FX (Post-K)	SPARC64 XIfx (PRIMEHPC FX100)
ISA (Base)	Armv8.2-A	SPARC-V9
ISA (Extension)	SVE	HPC-ACE2
Process Node	7nm	20nm
Peak Performance	>2.7TFLOPS	1.1TFLOPS
SIMD	512-bit	256-bit
# of Cores	48+4	32+2
Memory	HBM2	HMC
Memory Peak B/W	1024GB/s	240GB/s x2 (in/out)



NVIDIA GH200 Grace Hopper Superchip

https://resources.nvidia.com/en-us-grace-cpu/grace-hopper-superchip

Modernising a Code for SIMD?

How to get SIMD vectorisation?

1. Write "nice and simple" for loops and hope that the compiler is smart
    a. If it's not smart enough, try a different compiler
    b. Try -ffast-math if the code supports it
    c. Use OpenMP directives

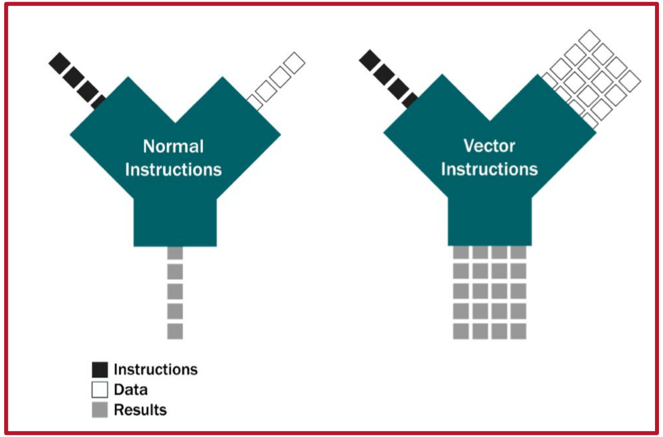
    See some vectorisation examples in C++ or in Fortran

2. Use built-in vector types (gcc, clang) → Madgraph talk later
3. Use vector abstraction libraries, e.g. VecCore
4. Direct low-level vector programming using intrinsics (e.g. Intel Intrinsics Guide)

Complexity

Vectorisation Requirements

- Every lane of a vector register must run the same instruction
  - Simple branches are possible, though
- Number of loop iterations should be computable upfront
- Data should be contiguous in memory: arrays, vectors, spans
- Pitfalls:
  - Unknown / non-inlinable / virtual functions in loop body
  - Loop-carried dependencies
  - Side effects
  - Exceptions (including floating-point exceptions)
  - Pointer aliasing
  - Register pressure, …



Paradigm shift: Data-oriented design

- Dynamically allocating memory breaks vectorisation
  - On GPUs, it is possible but prohibitively slow
- It is better to have pre-allocated, contiguous arrays, through which the algorithm iterates efficiently
  - Critical for SIMD
  - Excellent for CPU caches and MMU
  - Most efficient way to load memory on GPU
- → Event generators should be converted to exploit data parallelism, and run with batches of events that are densely placed in memory

Typical code transformations

Single-event interface

```
void computeBatch(std::size_t N) {
  for (unsigned int i = 0; i < N; ++i) {
    FourVector in1 = generateParticle(randomGenerator);
    FourVector in2 = generateParticle(randomGenerator);

    FourVector [out1, out2] = generateFinalState(in1, in2,
                                randomGenerator);

    double w1 = computePdfWeight(in1, in2);
    double w2 = computeMatrixElement(in1, in2,
                                     out1, out2);




    storeEvent(in1, in2, out1, out2, w1 * w2);
  }
}
```

Multi-event interface

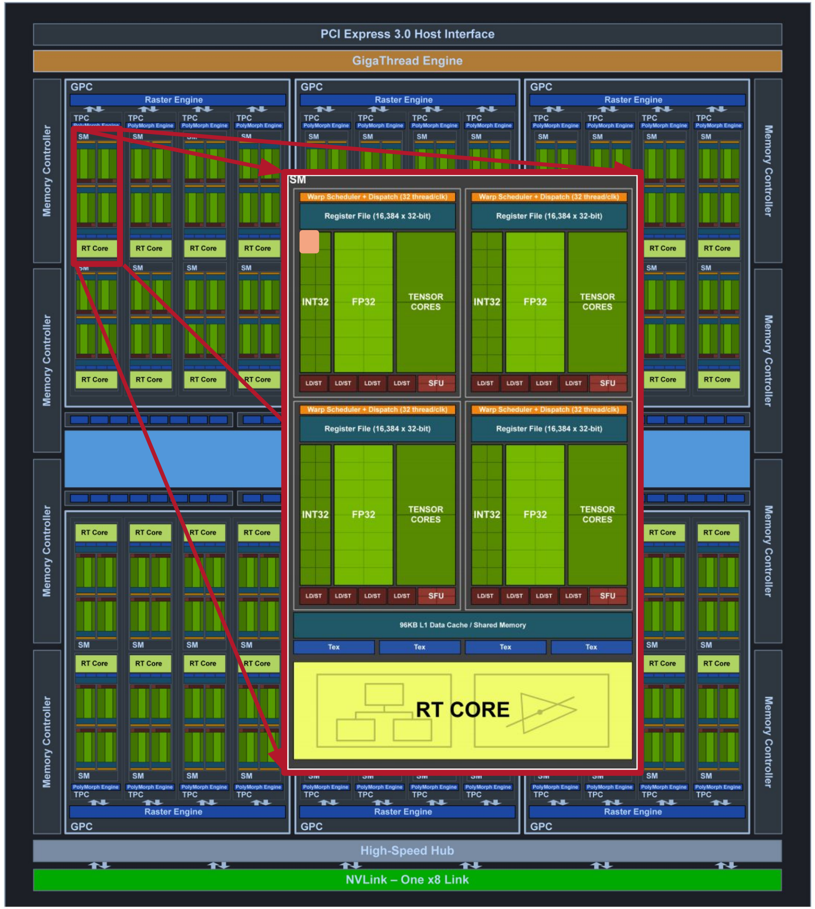
```
void computeBatch(std::size_t N) {

  array<FourVector,2*N> in = generateParticles(2, N,
                               randomGenerator);

  array<FourVector,2*N> out = generateFinalState(in,
                                randomGenerator);

  array<double, N> w1 = computePdfWeights(in);
  array<double, N> w2 = computeMatrixElements(in, out);

  array<double, N> finalWeight;
  for (unsigned int i = 0; i < N; ++i) {
    finalWeight[i] = w1[i] * w2[i];
  }

  storeEvents(in, out, finalWeight);
}
```

Modernising a Code for GPUs?

Anatomy of a GPU

- GPUs consist of many simple processors ("SMs", "Stream processors")
- Each SM runs SIMD-like instructions
- Work is sent as a grid of "threads" and "blocks"
- ComputeKernel<<<1,1>>>();

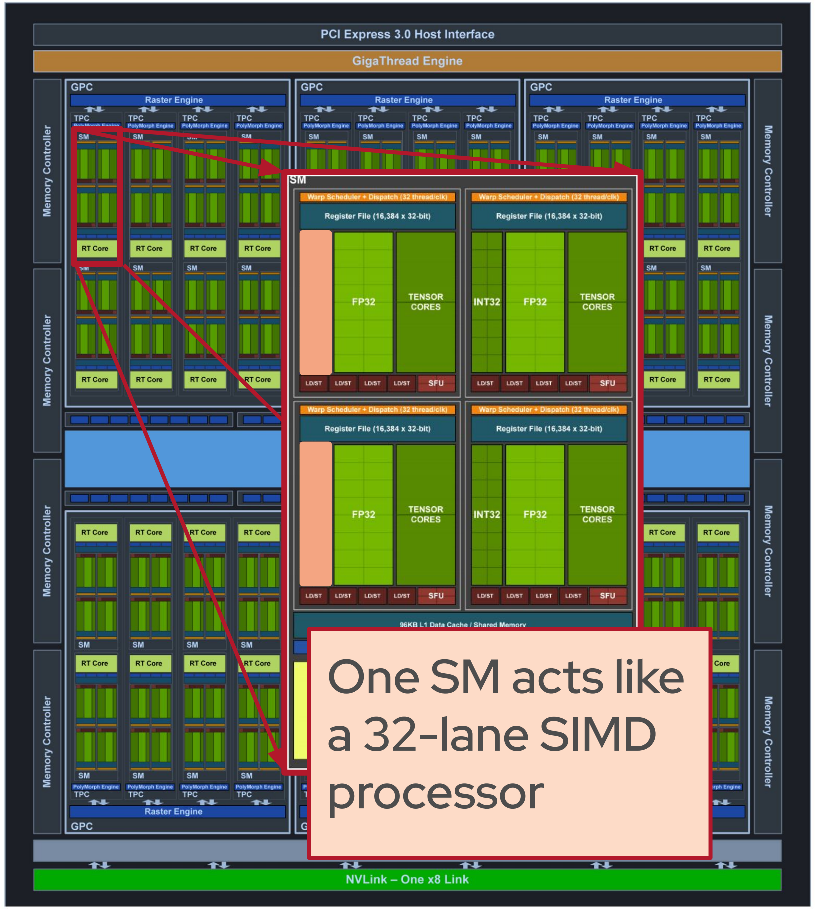
Nvidia Tesla T4



What happens on the GPU

- GPUs consist of many simple processors ("SMs", "Stream processors")
- Each SM runs SIMD-like instructions
- Work is sent as a grid of "threads" and "blocks"
- ComputeKernel<<< 2, 32>>>();

Nvidia Tesla T4


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
Raster Engine
TPC
PolyMorph Engine
SM
RT Core
Memory Controller
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32
INT32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
NVLink – One x8 Link

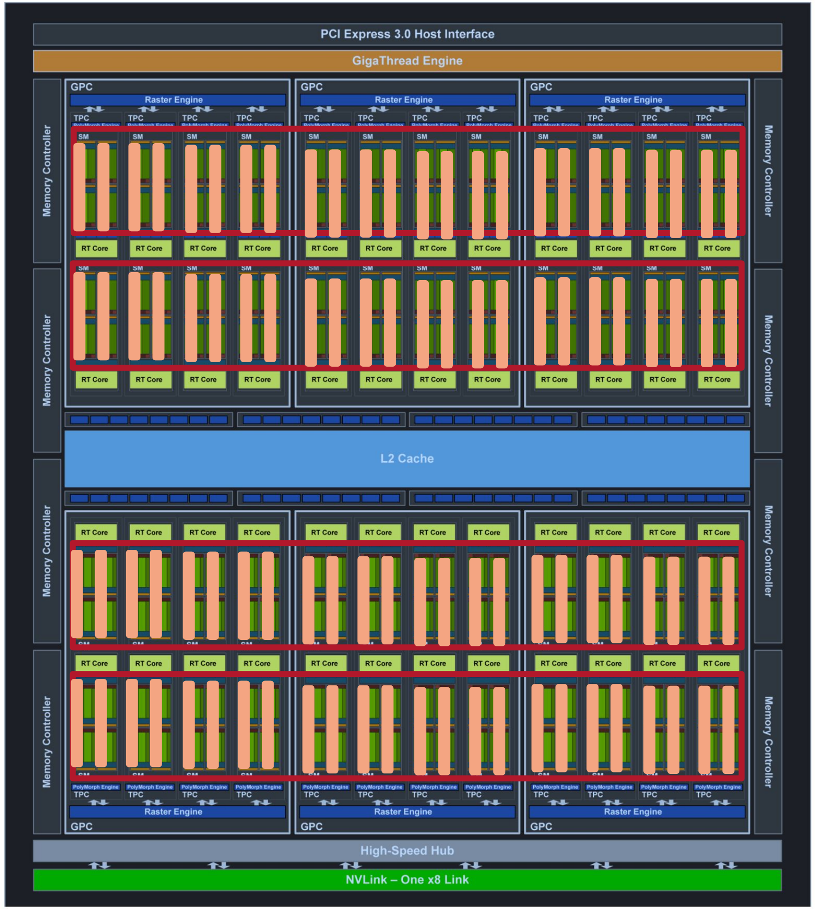
One SM acts like a 32-lane SIMD processor

Anatomy of a GPU

- ComputeKernel<<< 40 , 64 >>>();
- Even on the relatively simple Tesla T4, we need **2560 threads** to have "one full wave" of warps that span across the entire GPU
- For modern GPUs, we are looking at ~ **13'312** (AMD MI250)
- → Event generators should work on **large batches of data**

  Large ~> 10 000

Nvidia Tesla T4



~~Vectorisation~~ GPU Requirements

- Every ~~lane of a vector register~~ *thread in a warp* must run the same instruction
  - Simple *and larger but expensive* branches are possible, though
- Number of loop iterations should be computable upfront *to be fast*
- Data should be contiguous in memory *to load fast*
- Pitfalls:
  - Unknown / non-inlinable / virtual functions in loop body *slow the GPU down drastically*
  - Loop-carried dependencies *require slow synchronisation*
  - Side effects *require slow synchronisation*
  - Exceptions (including floating-point exceptions) *are not supported*
  - Pointer aliasing *might create race conditions / leads to slow memory access*
  - Register pressure *slows the GPU down*

~~Vectorisation~~ GPU Requirements

- Every ~~lane of a vector register~~ *thread in a warp* must run the same instruction
  - Simple *and larger but expensive* branches are possible, though
- Number of loop iterations should be computable upfront *to be fast*
- Data should be contiguous in memory *to load fast*
- Pitfalls:
  - Unknown / non-inlinable / virtual functions in loop body *slow the G*
  - Loop-carried dependencies *require slow synchronisation*
  - Side effects *require slow synchronisation*
  - Exceptions (including floating-point exceptions) *are not supported*
  - Pointer aliasing *might create race conditions / leads to slow memory access*
  - Register pressure *slows the GPU down*

Code requirements for SIMD and GPUs are largely the same!

Typical code transformations

Multi-event interface (CPU)

```
void computeBatch(std::size_t N) {

  array<FourVector,2*N> in = generateParticles(2, N,
                               randomGenerator);

  array<FourVector,2*N> out = generateFinalState(in,
                                randomGenerator);


  array<double, N> w1 = computePdfWeights(in);
  array<double, N> w2 = computeMatrixElements(in, out);


  array<double, N> finalWeight;
  for (unsigned int i = 0; i < N; ++i) {
    finalWeight[i] = w1[i] * w2[i];
  }

  storeEvents(in, out, finalWeight);
}
```

Multi-event interface (GPU)

```
void computeBatch(std::size_t N) {
  allocateCPU(N, in, out, w2);
  allocateGPU(N, inGPU, outGPU, w2GPU);
  generateParticles<<<B,T>>>(N, inGPU, randomGenerator);

  generateFinalState<<<B,T>>>(inGPU, outGPU,
                              randomGenerator);
  copyToHost(inGPU, in); copyToHost(outGPU, out);

  array<double, N> w1 = computePdfWeights(in);
  computeMatrixElements<<<B,T>>>(inGPU, outGPU, w2GPU);
  copyToHost(w2GPU, w2);

  array<double, N> finalWeight;
  for (unsigned int i = 0; i < N; ++i) {
    finalWeight[i] = w1[i] * w2[i];
  }

  storeEvents(in, out, finalWeight);
}
```

Typical code transformations

Multi-event interface
(CPU)

More complicated
memory management

Multi-event interface
(GPU)

```
void computeBatch(std::size_t N) {

  array<FourVector,2*N> in = generateParticles(2, N,
                                randomGenerator);

  array<FourVector,2*N> out = generateFi
                                 randomGe


  array<double, N> w1 = computePdfWeights(in);
  array<double, N> w2 = computeMatrixElements(in, out);


  array<double, N> finalWeight
  for (unsigned int i = 0; i <
    finalWeight[i] = w1[i] * w
  }

  storeEvents(in, out, finalWeight);
}
```

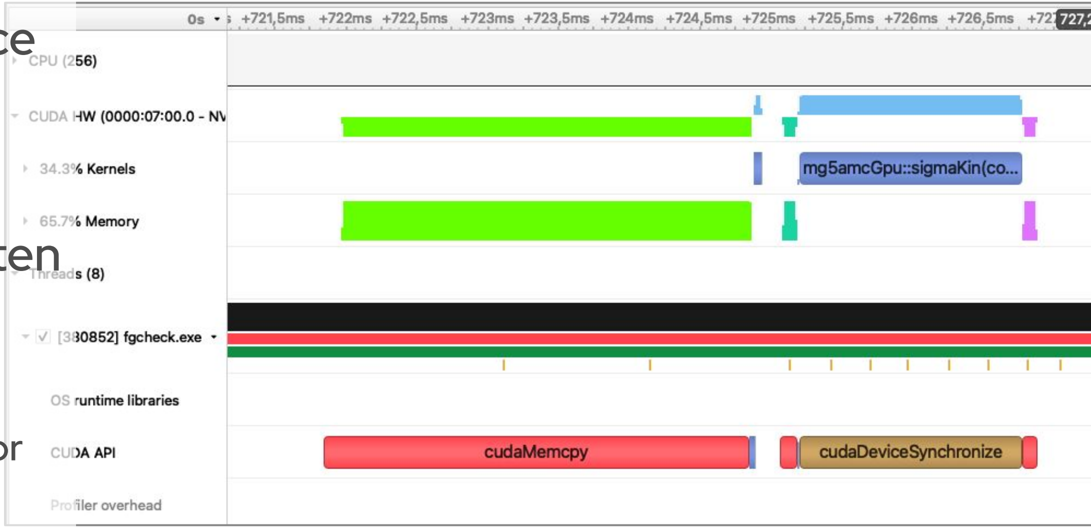
Parallel CPU and GPU
execution

Deal with copy latencies
and data dependencies;
synchronisation!

```
void computeBatch(std::size_t N) {
  allocateCPU(N, in, out, w2);
  allocateGPU(N, inGPU, outGPU, w2GPU);
  generateParticles<<<B,T>>>(N, inGPU, randomGenerator);

       ateFinalState<<<B,T>>>(inGPU, outGPU,
                               randomGenerator);
       oHost(inGPU, in); copyToHost(outGPU, out);

  array<double, N> w1 = computePdfWeights(in);
  computeMatrixElements<<<B,T>>>(inGPU, outGPU, w2GPU);
  copyToHost(w2GPU, w2);

  array<double, N> finalWeight;
  for (unsigned int i = 0; i < N; ++i) {
    finalWeight[i] = w1[i] * w2[i];
  }

  storeEvents(in, out, finalWeight);
}
```

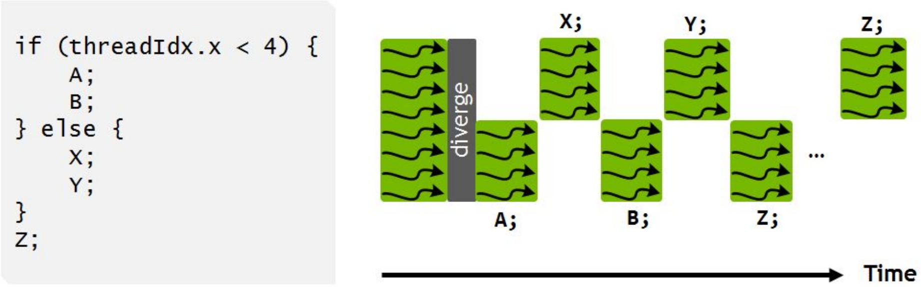
Copy latencies and GPUs

- When using GPUs, data must be copied between host and device
- **Compute work >> copy work** required for good speedups
- For event generators, this is often the case!
  - Copy four-momenta
  - Compute hundreds of diagrams for thousands of events
  - Copy matrix elements



Branches

- If all threads in a warp have to run the same instruction, how do we branch?
  - The warp is split
  - Threads that don't take the branch are disabled
    → Throughput reduced
- Branching is not the fastest way of using a GPU, but it's OK on modern GPUs
- But event generators don't branch that often ...



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

developer.nvidia.com/blog/inside-volta/

Vectorisation and GPU wrap up

- The code transformations necessary for SIMD and GPU usage are very similar
  - **Single → Multi-event interface**
  - Generate events in batches; **make N large**
  - Explicit memory management, careful placement of data in arrays
  - + for GPU: Copying of data host ← → device where necessary
- The most efficient way to get there?

  A team of:
  - Experts who know the workflow and data dependencies of an event generator
  - Experts with experience in SIMD and GPU programming

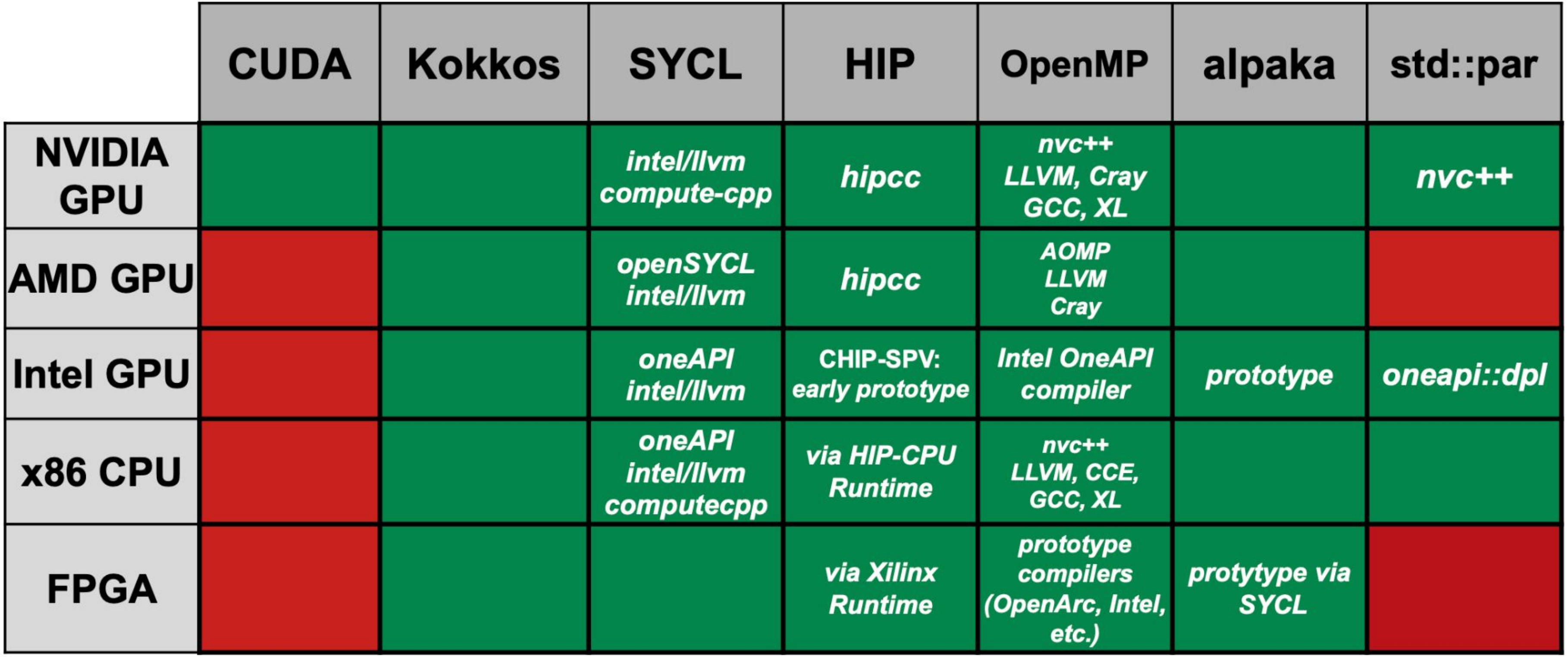
Targeting different hardware?

- Nvidia's CUDA (2007)
  - Extensive documentation, tools, online resources
  - Exclusively for Nvidia GPUs
- AMD's ROCm (HIP, 2016)
  - Similar to CUDA, but targets AMD GPUs
- Intel GPUs with SYCL
- Abstraction frameworks:
  - Program abstract kernels, translate to native GPU language later
  - Target multiple GPU architectures *and* CPUs
  - How to choose between
    OpenCL, SYCL, Alpaka, Kokkos, OpenMP, std::par?

Portable GPU codes?

C. Leggett CHEP23

Fortran ↓



	CUDA	Kokkos	SYCL	HIP	OpenMP	alpaka	std::par
NVIDIA GPU	🟩	🟩	🟩 intel/llvm compute-cpp	🟩 hipcc	🟩 nvc++ LLVM, Cray GCC, XL	🟩	🟩 nvc++
AMD GPU	🟥	🟩	🟩 openSYCL intel/llvm	🟩 hipcc	🟩 AOMP LLVM Cray	🟩	🟥
Intel GPU	🟥	🟩	🟩 oneAPI intel/llvm	🟩 CHIP-SPV: early prototype	🟩 Intel OneAPI compiler	🟩 prototype	🟩 oneapi::dpl
x86 CPU	🟥	🟩	🟩 oneAPI intel/llvm computecpp	🟩 via HIP-CPU Runtime	🟩 nvc++ LLVM, CCE, GCC, XL	🟩	🟩
FPGA	🟥	🟩	🟩	🟩 via Xilinx Runtime	🟩 prototype compilers (OpenArc, Intel, etc.)	🟩 protytype via SYCL	🟥

Portable Parallelization Strategies: Recommendations

C. Leggett CHEP23

Software and hardware are still rapidly changing

- Lots of interactions with API developers in hackathons and to fix bugs
- Results remain preliminary

API recommendations are very application dependent

- All perform approximately equally for simple kernels
- Complex algorithms and chained kernels bring out weaknesses of all APIs
  - interaction with external libraries adds extra complexities
  - even compilation can be an issue

Learning curve / language complexity of APIs not all the same

- std::par → OMP → Kokkos / SYCL → alpaka → OMP
- subjective and dependent on code complexity and previous experience

Porting from Serial CPU code → GPU concepts is the biggest hurdle

- starting with optimized code is extra challenging

Very hard to extrapolate to next five years or beyond

- Vendors are pushing in different directions (but towards standards)
- Increasing proximity of CPU / GPU / memory will have significant impact

So is it worth the effort?

- The reengineering is a lot of work ...

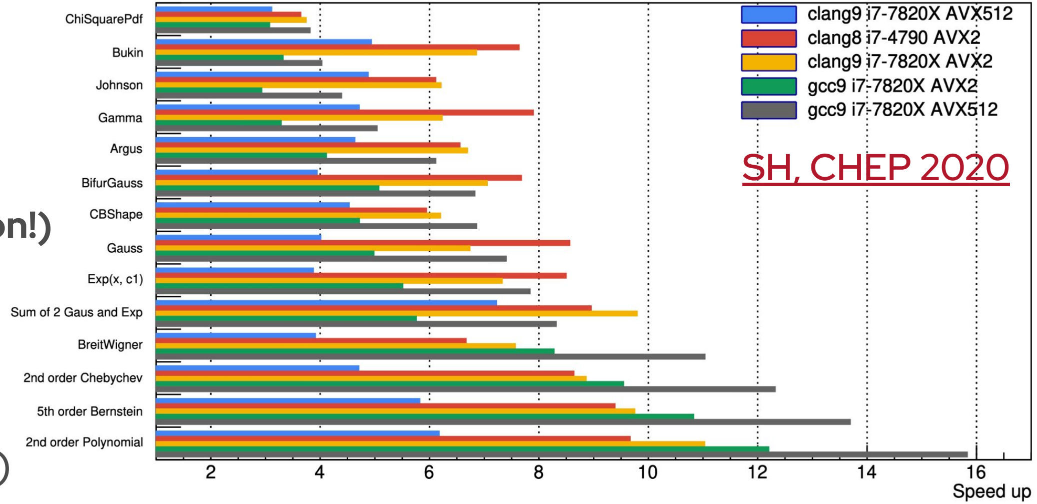
But:

- Collaborations that might have looked "unlikely" have been formed successfully between different communities
- Old cold gets revisited, new ideas might pop up
- Better adaptation to new hardware
  - Saving electricity, CO2, hardware
  - Start using the unused SIMD slots of our CPUs
  - Offloading to GPUs is in reach
  - Abstraction frameworks will allow us to adapt to changing hardware landscape

Some real-life examples

Example 1: RooFit

- Converting ROOT's RooFit from single to multi-event workflow:
  - 1 year fellow at 30-40% time
  - **3x speedup (without vectorisation!)**
- Making code autovectorisation friendly:
  - 1 summer student
  - **3x → 16x speedup (double precision!)**
- GPU offloading with CUDA:
  - 1 technical student
  - **25x - 40x on a gaming GPU** (low double-precision performance)

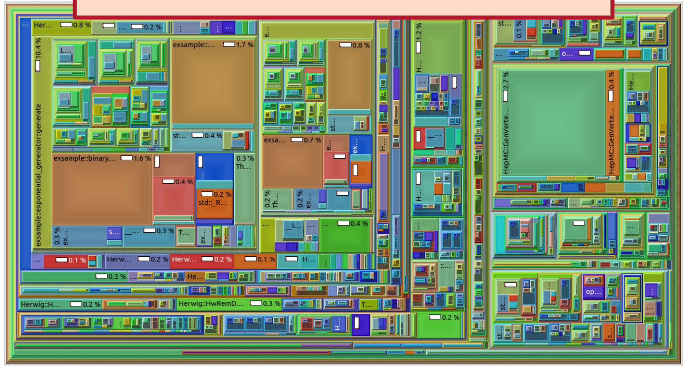


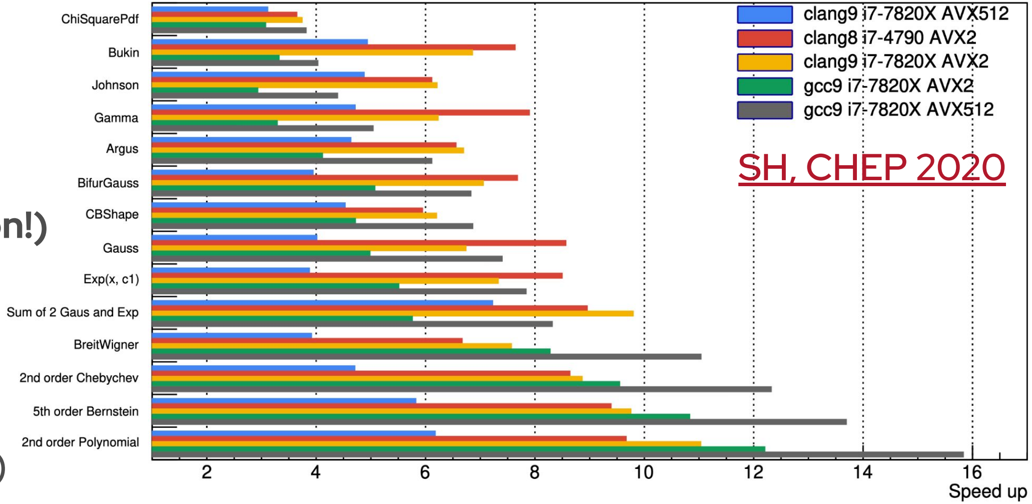
SH, CHEP 2020

Example 1: RooFit

- Converting ROOT's RooFit from single to multi-event workflow:
  - 1 year fellow at 30-40% time
  - **3x speedup (without vectorisation!)**
- Making code autovectorisation friendly:
  - 1 summer student
  - **3x → 16x speedup (double precision!)**
- GPU offloading with CUDA:
  - 1 technical student
  - **25x - 40x on a gaming GPU** (low double-precision performance)

Remember this from James' talk?



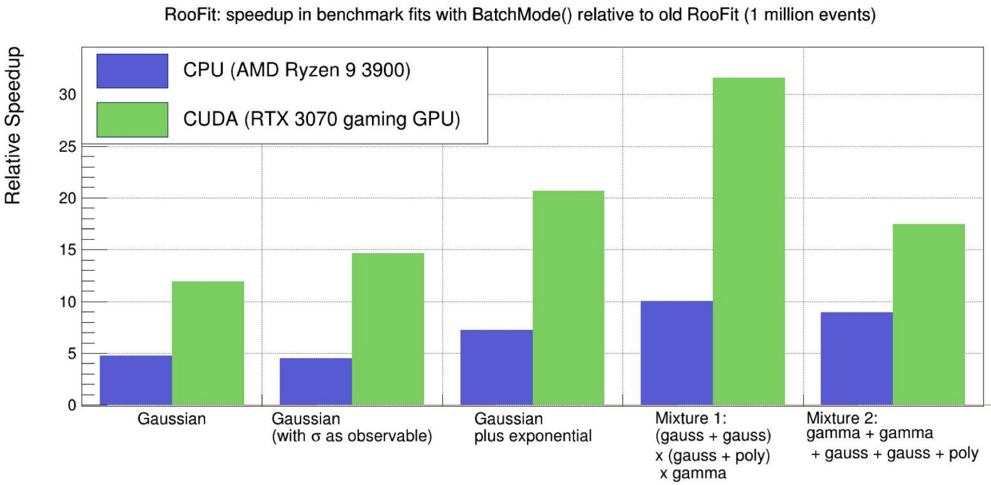


SH, CHEP 2020

Example 1: RooFit

- Converting ROOT's RooFit from single to multi-event workflow:
  - 1 year fellow at 30-40% time
  - **3x speedup (without vectorisation!)**
- Making code autovectorisation friendly:
  - 1 summer student
  - **3x → 16x speedup (double precision!)**
- GPU offloading with CUDA:
  - 1 technical student
  - **25x - 40x on a gaming GPU** (low double-precision performance)

J Rembser, CHEP 2023



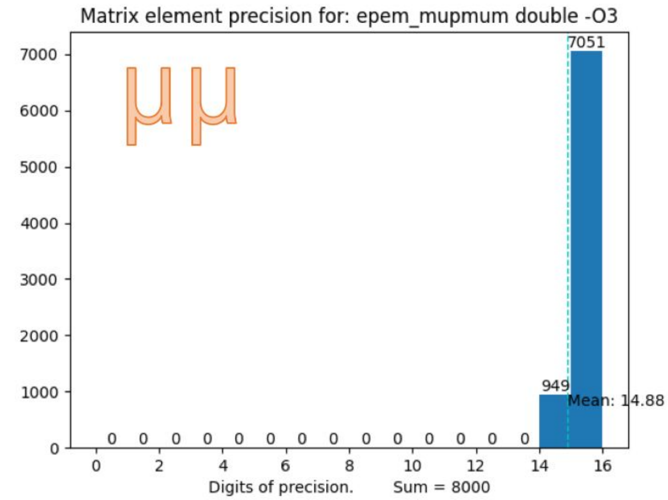
See real-life example with an EventGenerator:
Madgraph talk today or
Sherpa talk today

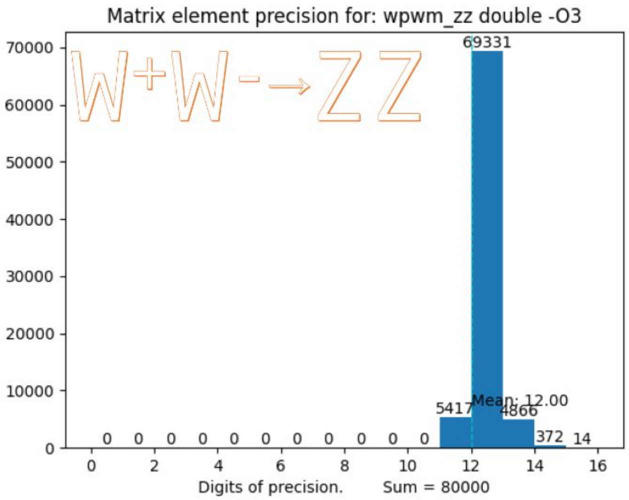
Example 2: Can Madgraph run in single precision?

- Computers may lie about precision of floating-point numbers
- CADNA is a library with special floating-point types to measure precision and instabilities in C++ and Fortran
- See seminar at CERN
- For madgraph, we were wondering if single precision works

$$P(x,y)= 9x^4 - y^4 + 2y^2$$

```
Without CADNA:
P(10864,18817) = 2.0000000000000000 (exact value: 1)
P(1/3,2/3) = 0.8024691358024691

With CADNA:
P(10864,18817) = @.0 (exact value: 1)
P(1/3,2/3) = 0.802469135802469E+000
---------------------------------------------


0 UNSTABLE DIVISION(S)
0 UNSTABLE POWER FUNCTION(S)
0 UNSTABLE MULTIPLICATION(S)
0 UNSTABLE BRANCHING(S)
0 UNSTABLE MATHEMATICAL FUNCTION(S)
0 UNSTABLE INTRINSIC FUNCTION(S)
2 UNSTABLE CANCELLATION(S)
```


Matrix element precision for: epem_mupmum double -O3


Matrix element precision for: wpwm_zz double -O3

Double-precision MEs

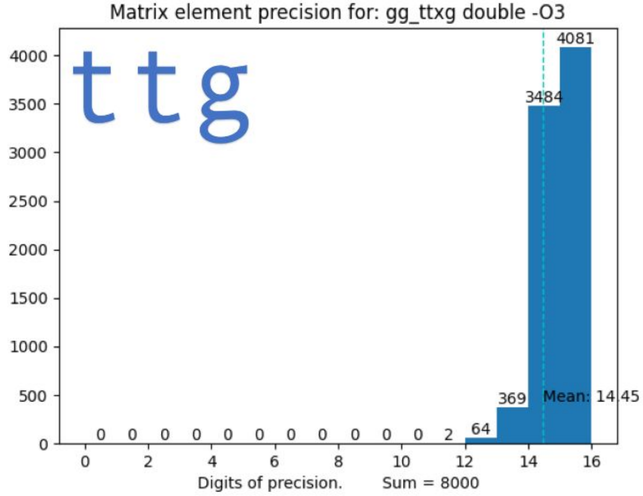
e+e- → μ+μ-

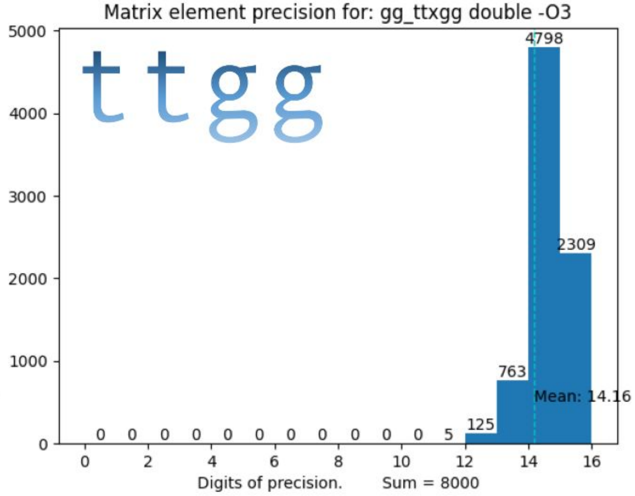
W+W- → ZZ

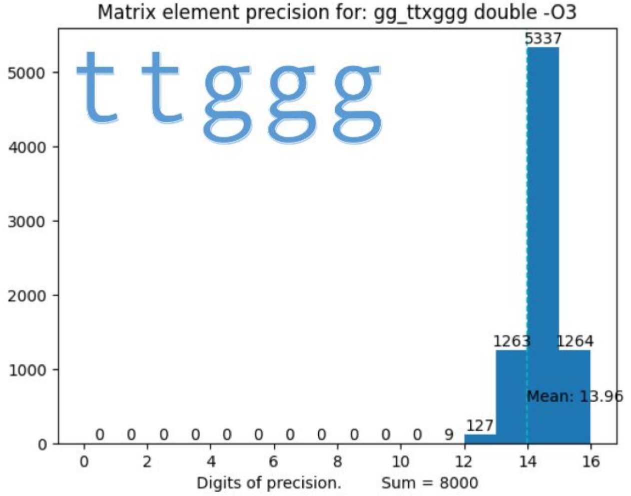
gg → ttg

gg → ttgg

gg → ttggg


Matrix element precision for: gg_ttxg double -O3

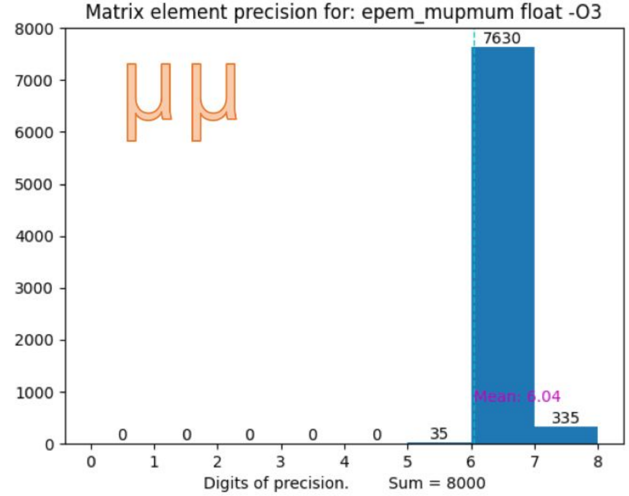

Matrix element precision for: gg_ttxgg double -O3

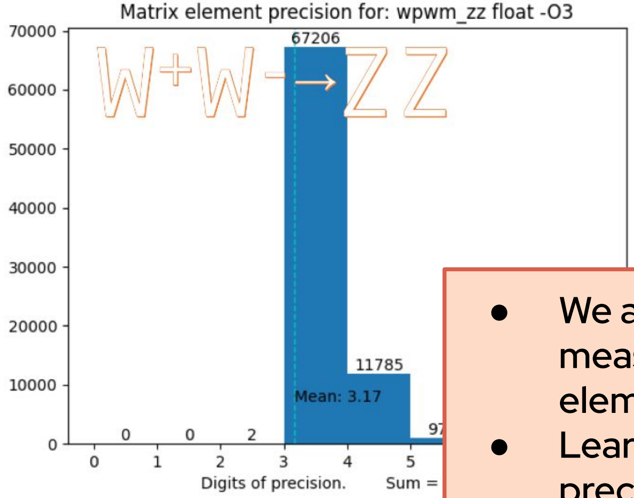

Matrix element precision for: gg_ttxggg double -O3

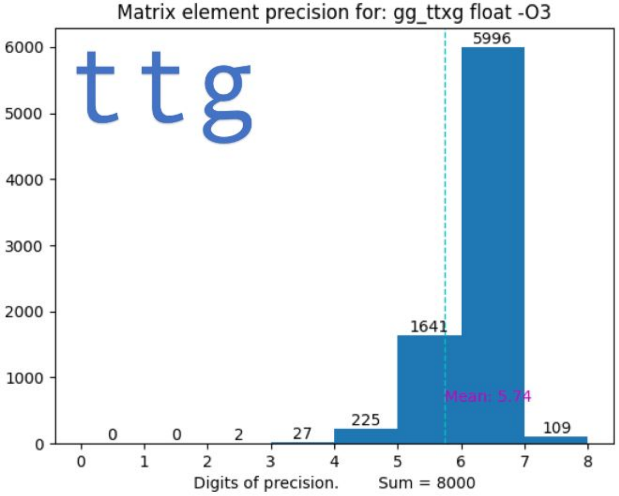
Single-precision MEs

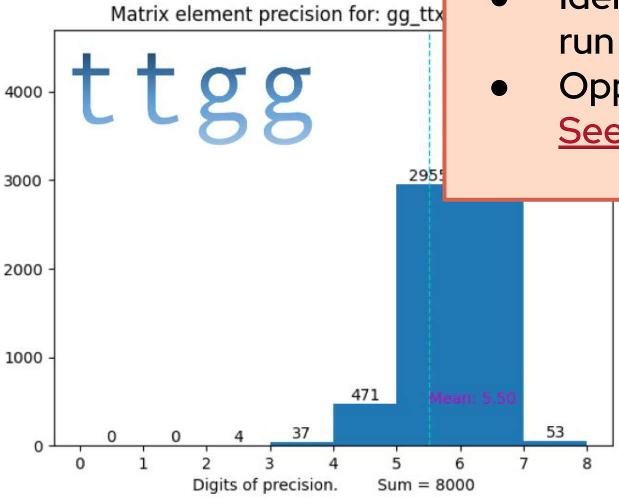
e+e- → μ+μ-

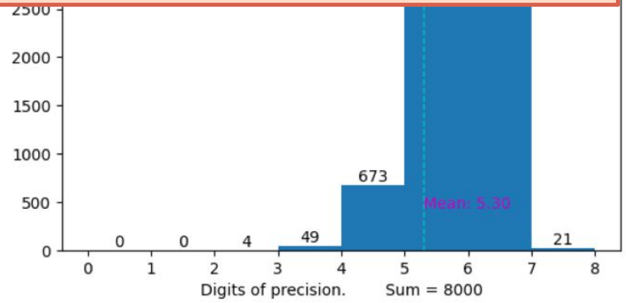
- We applied CADNA to madgraph to measure the precision of matrix elements
- Learned that generally double precision is needed
- Identified some computations that can run in single precision, though!
- Opportunity for speedup
  See Madgraph meeting











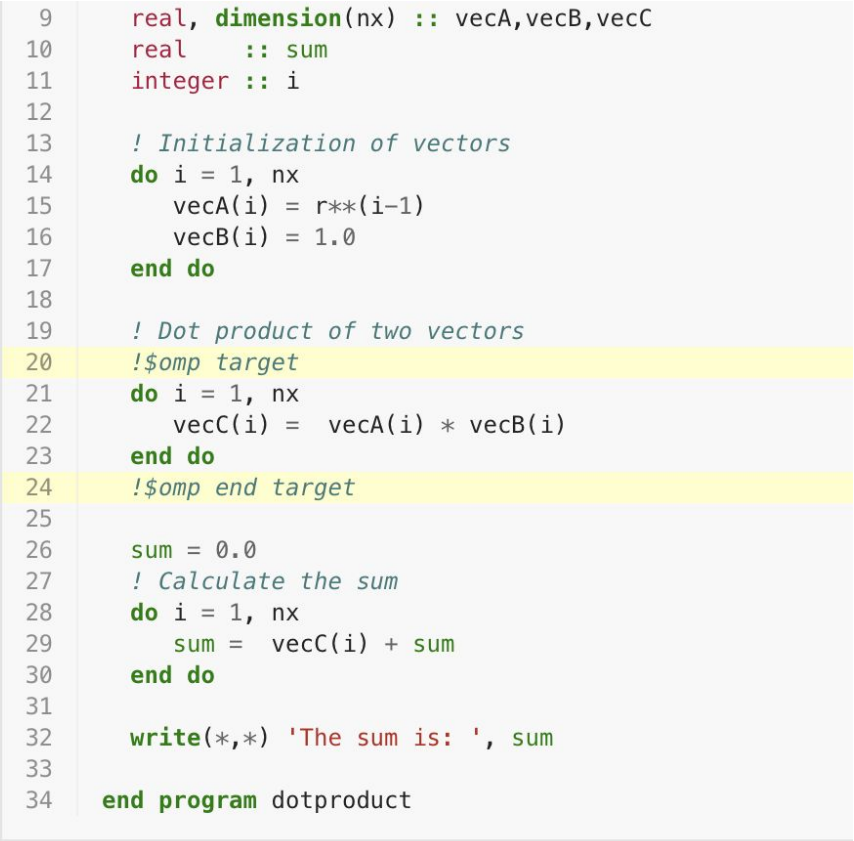
Closing Remarks

- If you look for speedup opportunities, you find something
- "New" hardware is around since 2013 (AVX2); we should be using it efficiently
- Event generators are in a particularly good situation, as multi-event data parallelism caters perfectly to SIMD and GPUs
- By converting to multi-event batched workflows, we get ourselves in a good position for what may come in terms of hardware evolution
- Double-precision performance of GPUs might be an issue (expensive HW)
- The opportunities outweigh the challenges **if both generator experts and performance-oriented people are allowed to prioritise such work**

Backup

On OpenMP offloading in Fortran

- GPU offloading of a heavy loop can be trivial with OpenMP > 4.5
  - Both in C++ and Fortran
- It might not be the most efficient (implicit data transfers)
- SIMD for CPU version is similarly trivial: `!$omp simd`



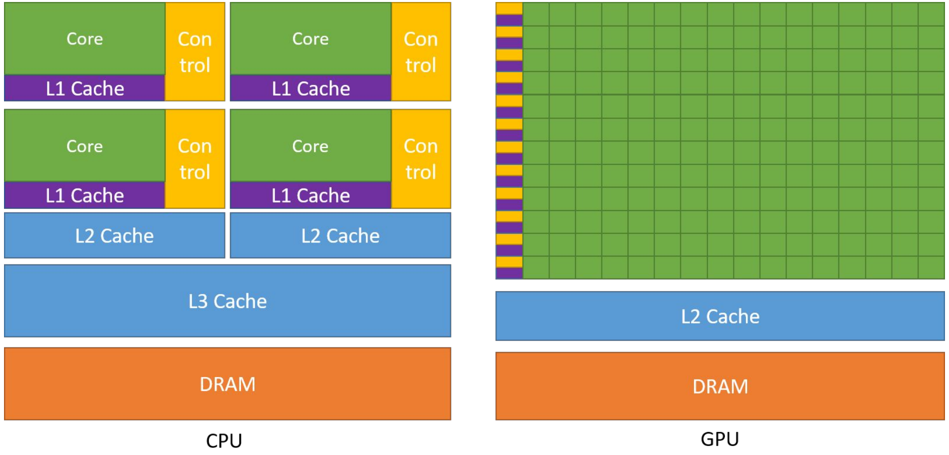
```fortran
  real, dimension(nx) :: vecA,vecB,vecC
  real    :: sum
  integer :: i

  ! Initialization of vectors
  do i = 1, nx
     vecA(i) = r**(i-1)
     vecB(i) = 1.0
  end do

  ! Dot product of two vectors
  !$omp target
  do i = 1, nx
     vecC(i) =  vecA(i) * vecB(i)
  end do
  !$omp end target

  sum = 0.0
  ! Calculate the sum
  do i = 1, nx
     sum =  vecC(i) + sum
  end do

  write(*,*) 'The sum is: ', sum

end program dotproduct
```

https://enccs.github.io/openmp-gpu/target/

CPU vs GPU

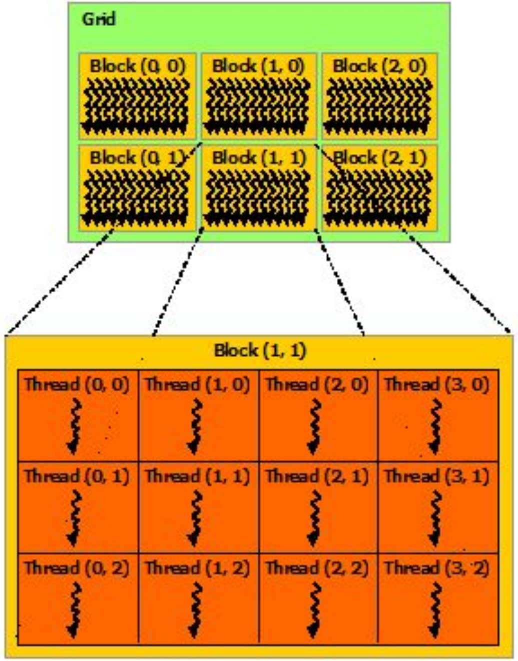
- CPUs:
  - Try to reduce latency of execution
  - Every core (~ thread) has its own cache and control unit, several levels of caches
  - Lots of energy and hardware spent on latency reduction
- GPUs:
  - Threads **share** control units and a lot of cache space
  - More transistors devoted to data processing
  - → Massive data parallelism
  - → Higher latency

https://docs.nvidia.com/cuda/cuda-c-programming-guide/



CUDA Execution Model

- **Kernel**: Function invoked on the GPU
- **Grid**: Set of blocks running for a specific kernel
- **Block**: Set of threads on the same "streaming multiprocessor"
- **Thread**: Set of instructions to be executed

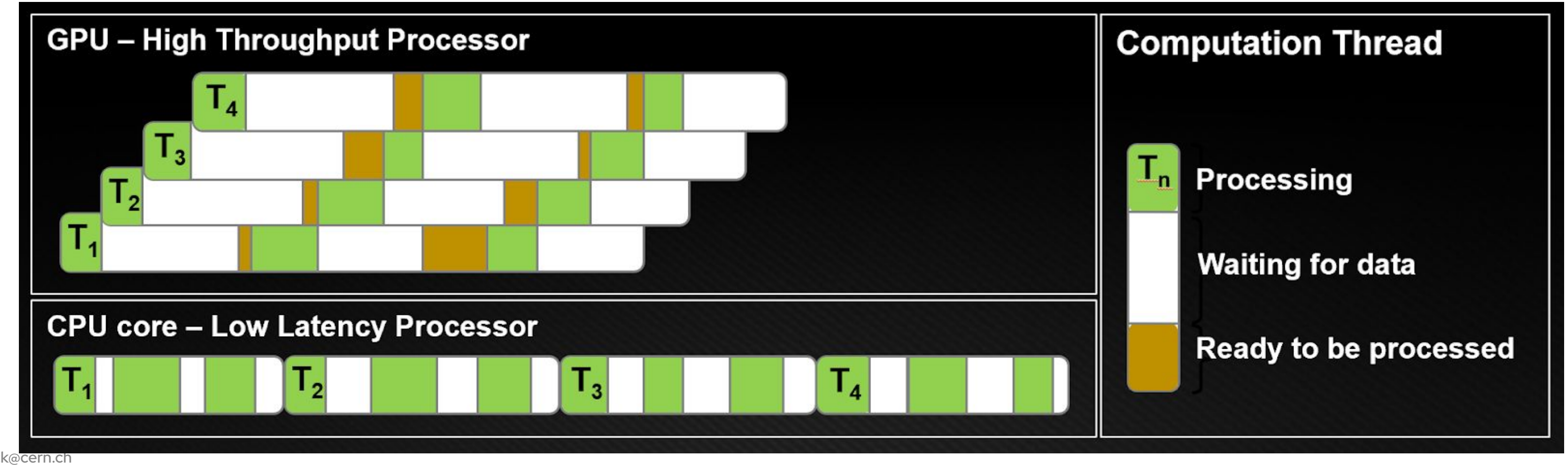


Memory Latency and GPUs

- If a warp stalls, other warps step in
  - Requires multiple warps / SM
- Recommendation: Start with 256 threads – 8 warps – and try other numbers

```
for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
```

- CPUs minimise latency, GPUs hide it
- You need enough work for the GPU to do this successfully

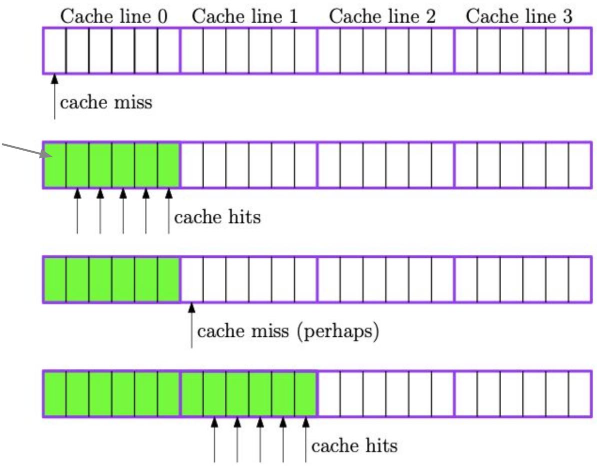


developer.nvidia.com/blog/cuda-refresher-reviewing-the-origins-of-gpu-computing/

Efficient memory access

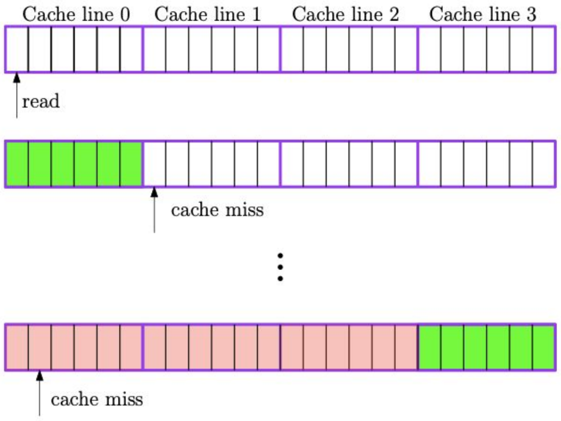
```
// Access with stride 1
__global__
void k_copy(double* x, double* y, int n) {
 for(int i = threadIdx.x + blockDim.x*blockIdx.x;
   i < n; i += blockDim.x * gridDim.x)
  y[i] = x[i];
}
```

DRAM is read in bursts
(e.g. 16 float/int in GDDR6)



L. Einkemmer

```
// Access with larger stride
__global__
void k_copy(double* x, double* y, int n) {
 for(int i = threadIdx.x * gridDim.x + blockIdx.x;
   i < n; i += blockDim.x * gridDim.x)
  y[i] = x[i];
}
```

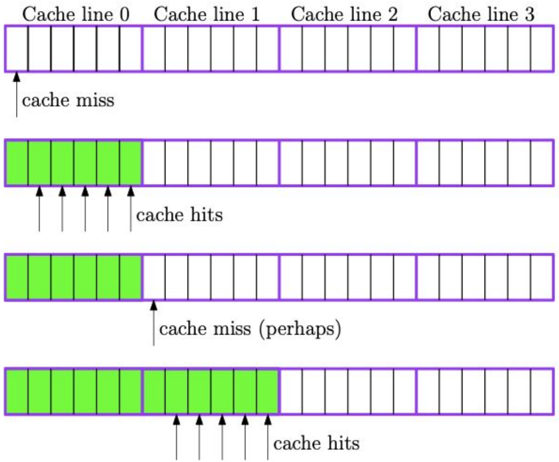


Efficient memory access

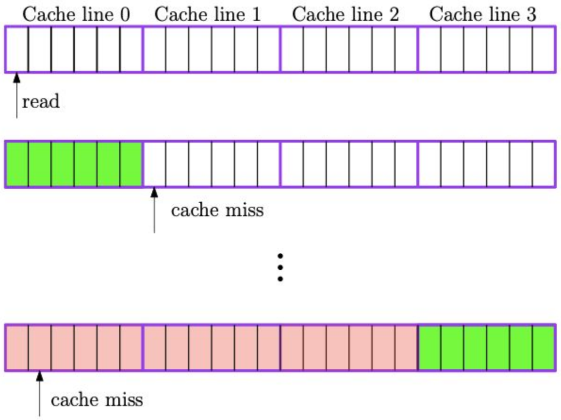
```
// Access with stride 1
__global__
void k_copy(double* x, double* y, int n) {
  for(int i = threadIdx.x + blockDim.x*blockIdx.x;
      i < n; i += blockDim.x * gridDim.x)
    y[i] = x[i];
}
```

Neighbouring threads should read neighbouring memory locations "coalesced access"

```
// Access with larger stride
__global__
void k_copy(double* x, double* y, int n) {
  for(int i = threadIdx.x * gridDim.x + blockIdx.x;
      i < n; i += blockDim.x * gridDim.x)
    y[i] = x[i];
}
```



L. Einkemmer


Cache line 0
Cache line 1
Cache line 2
Cache line 3
read
cache miss
cache miss