Four years of the MIC project at openlab – observations and conclusions

openlab Minor Review
February 26th 2013

Andrzej Nowak, CERN openlab
Based on the work of Sverre Jarp, Alfio Lazzaro, Julien Leduc, Andrzej Nowak, Klaus-Dieter Oertel, Liviu Valsan

CERN openlab

DISCLAIMER

Our tests are based on pre-production MIC / Xeon Phi hardware and software

History

Early access

- Work since MIC alpha (under RS-NDA)
- ISA reviews in 2008

Results

- 3 benchmarks ported from Xeon and delivering results: ROOT, Geant4, ALICE HLT trackfitter

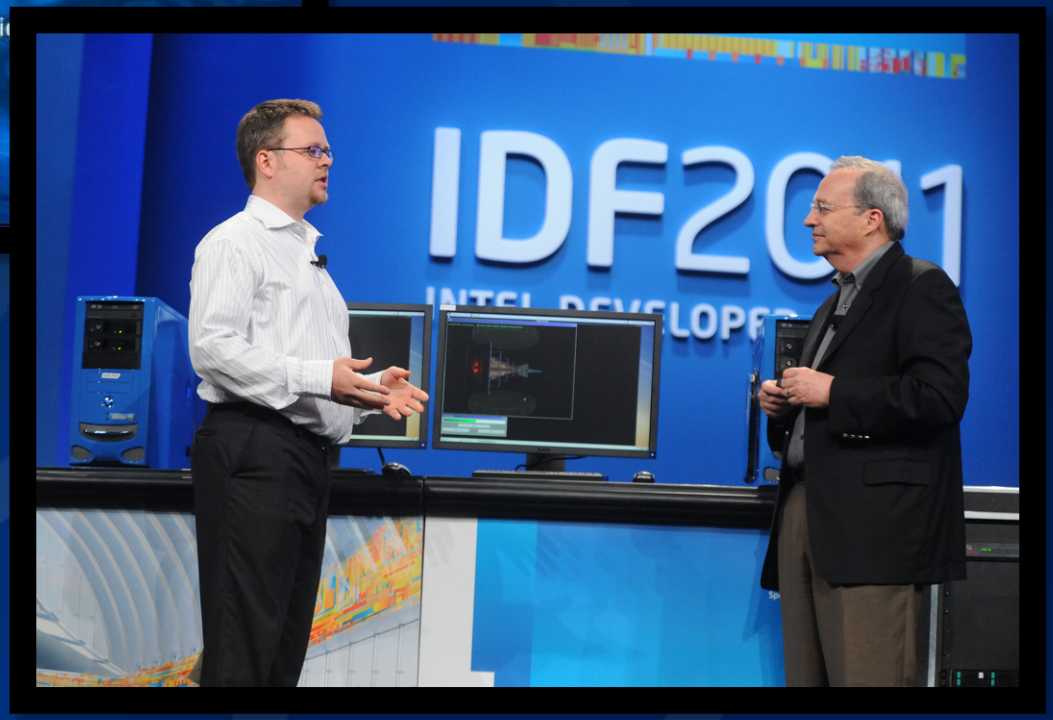
Expertise

- Understood and compared with Xeon
- **Post-launch dissemination**

openlab contributions to the MIC program

- **Ported and optimized 3 large representative benchmarks**
- **Feedback on the ISA**
- **Extensive feedback:**
  - performance and usability of SW/HW
  - possible future directions vis a vis HEP
  - All feedback registered and considered by Intel with tangible results
  - Had influence on the shape and format of future chips and software
- **Testimonials**
  - ISC'10 w/ Intel VP Kirk Skaugen
  - IDF'11 w/ Intel CTO Justin Rattner

openlab contributions to the MIC program



MIC hardware – Knights Corner

- **PCIe card form factor, a "PC in a PC"**
  - Linux OS on board
  - PCIe power envelope – <300W
  - Limited on-board memory (up to 8GB GDDR5)
  - >50 cores @ >1 GHz
- **x86 architecture**
  - 22nm process, 64-bit
  - P54C core: In-order, Superscalar
  - 8MB shared coherent cache
  - New ISA with new vector instructions
- **Floating point support through vector units**
  - 512-bit wide
- **4-wide hardware threading**
  - >200 threads in total
- **1 TFLOP DP**
  - Still with the programmability of a Xeon

KNF hardware architecture
(Xeon Phi prototype)

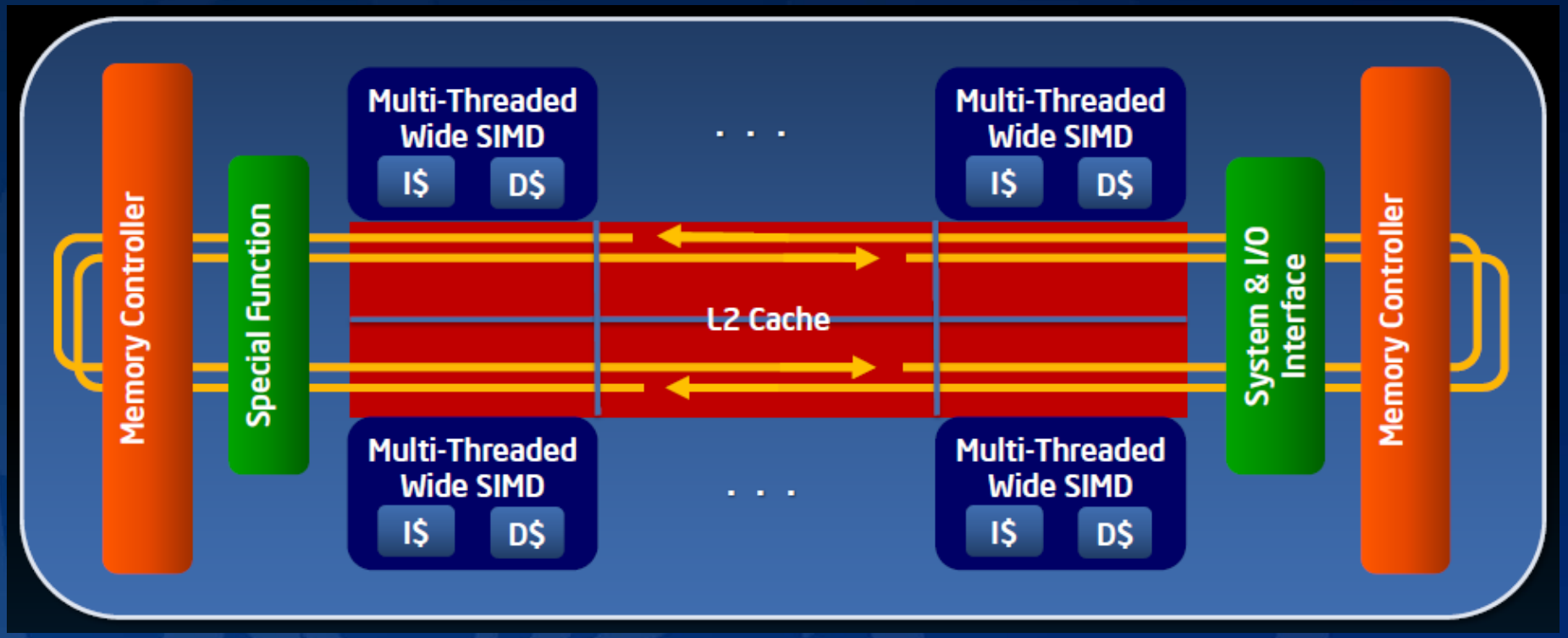
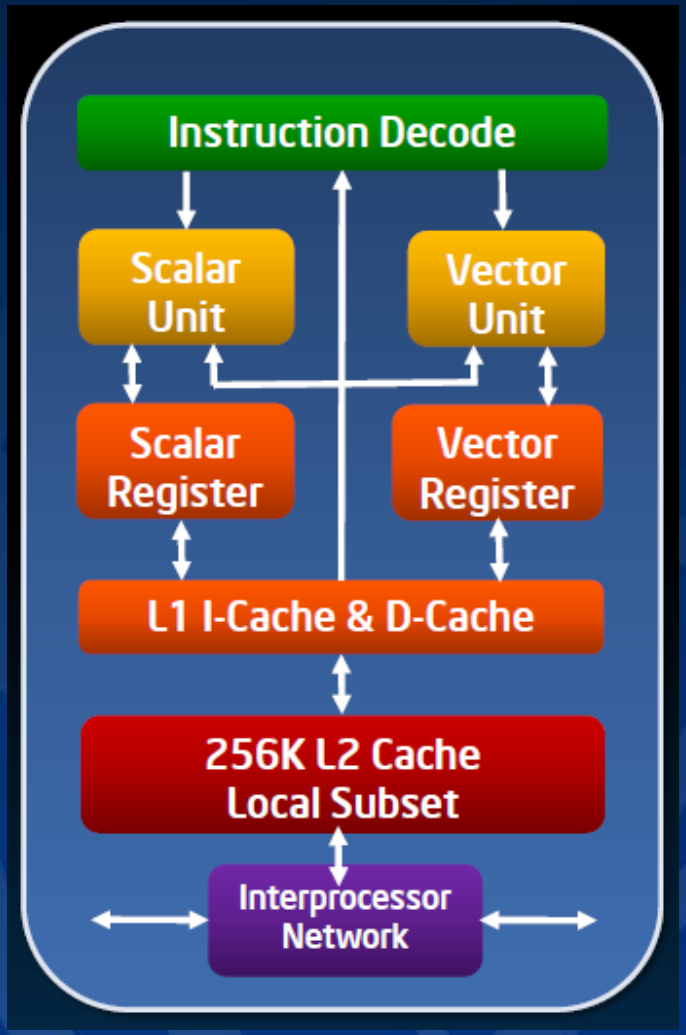


Image: Intel

Aubrey Isle core architecture
(KNF core)



- Superscalar
- 32k L1 I/D cache
- 256k Coherent L2 caches
- 2x512-bit ring bus inter-CPU network
- 4-wide SIMD with separate register sets

Image: Intel

MIC software

- **Extensive focus on programmability and interoperability with the Xeon**
  - PCI becomes transparent
- **Card OS is Linux**
- **Future GNU support possible**
- **Broad software support; standard Intel software available, same as on Xeon:**
  - C/C++/Fortran Compiler, VTune Amplifier profiler, various other
- **Various standard libraries supported**
- **Special APIs under development/consideration**
  - Host memory access
  - Low level communication
- **Various programming models considered**
- **“Familiar territory” approach – Intel wants to keep this “as x86 as possible”, with all the pros and cons**

Programmability (1)

Summing vector elements in C using OpenMP – openmp.org

```
#pragma omp parallel for reduction(+: s)
for (int i = 0; i < n; i++) {
  s += x[i];
}
```

Per element multiply in C++ using Intel® Array Building Blocks – intel.com/go/arbb

```
void products( const arbb::dense<arbb::f32>& a,
               const arbb::dense<arbb::f32>& b,
               arbb::dense<arbb::f32>& c) {
  c = a * b;
}
```

Dot product in Fortran using OpenMP – openmp.org

```
!$omp parallel do reduction ( + : adotb )
  do j = 1, n
    adotb = adotb + a(j) * b(j)
  end do
!$omp end parallel do
```

Sum in Fortran, using co-array feature – intel.com/software/products

```
REAL SUM[*]
CALL SYNC_ALL( WAIT=1 )
DO IMG= 2,NUM_IMAGES()
  IF (IMG==THIS_IMAGE()) THEN
    SUM = SUM + SUM[IMG-1]
  ENDIF
  CALL SYNC_ALL( WAIT=IMG )
ENDDO
```

Parallel function invocation in C using Intel® Cilk™ Plus – cilk.org

```
cilk_for (int i=0; i<n; ++i) {
  Foo(a[i]);
}
```

Parallel function invocation in C++ using Intel® Threading Building Blocks – threadingbuildingblocks.org

```
parallel_for (0, n,
  [=](int i) { Foo(a[i]); }
);
```

Programmability (2)

MPI code in C for clusters – intel.com/go/mpi

```
for (d=1; d<ntasks; d++) {
  rows = (d <= extra) ? avrow+1 : avrow;
  printf(" sending %d rows to task %d\n", rows, dest);
  MPI_Send(&offset, 1, MPI_INT, d, mtype, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, d, mtype, MPI_COMM_WORLD);
  MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, d, mtype, MPI_COMM_WORLD);
  MPI_Send(&b, NCA*NCB, MPI_DOUBLE, d, mtype, MPI_COMM_WORLD);
  offset = offset + rows;
}
```

Per element multiply in C using OpenCL – intel.com/go/opencl

```
kernel void
  dotprod( global const float *a,
           global const float *b,
           global float *c) {
    int myid = get_global_id(0);
    c[myid] = a[myid] * b[myid];
}
```

Matrix Multiply in Fortran using Intel® Math Kernel Library – intel.com/software/products

```
      call DGEMM(transa,transb,m,n,k,alpha,a,lda,b,ldb,beta,c,ldc)
```

Ported applications

- **ALICE/CBM track fitter prototype**
  - Threaded
  - Explicitly vectorized with a VC-like technology
- **MLFit**
  - Threaded (pthreads, MPI, OpenMP, TBB)
  - Vectorized (Cilk+)
- **Early multi-threaded Geant4 prototype**
  - Threaded (pthreads)
  - No vectorization
- **Test hardware**
  - Pre-production Knights Corner – 61 cores @ 1.1 GHz
  - Sandy Bridge-EP – 16 cores @ 2.7 GHz, Turbo on
  - Frequency unscaled results reported (1:1 comparison)

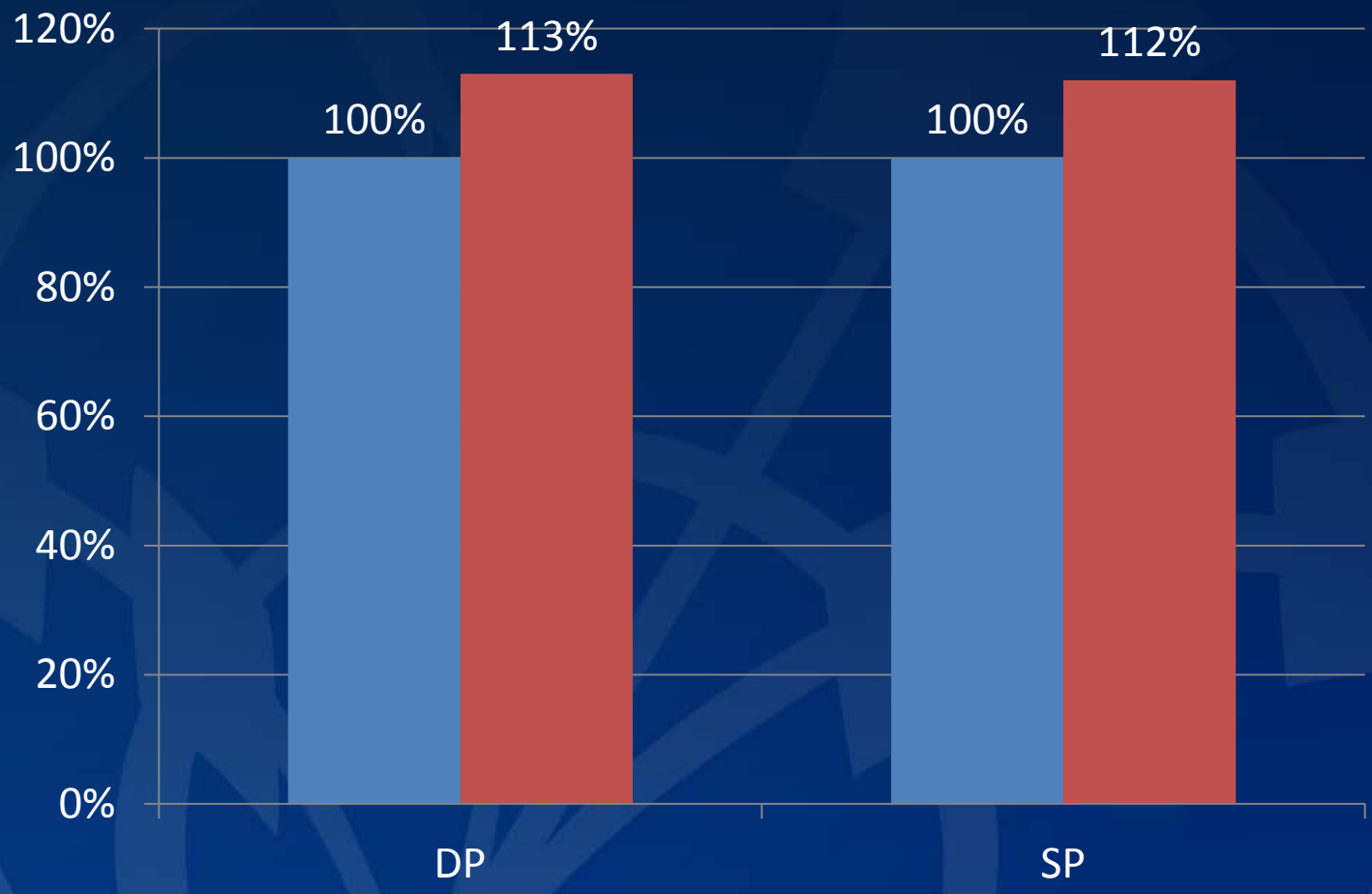
Porting – how much work?

	LOC	1st port time	New ports	Tuning
TF	< 1’000	days	N/A	2 weeks
MLFit	3’000	< 1 day	< 1 day	weeks
MTG	2’000’000	1 month	< 1 day	< 1 week

Porting - observations

- Ideal situation: just add a compiler switch and recompile
- Less-than-ideal: minor adaptations, including GCC/ICC differences if any + above step
- More likely: write parallel code or parallelize existing code + above steps
- Numerous libraries available: OpenMP, MPI, TBB, Cilk, MKL etc
- Vectorization (data parallelism) is key to achieve full performance

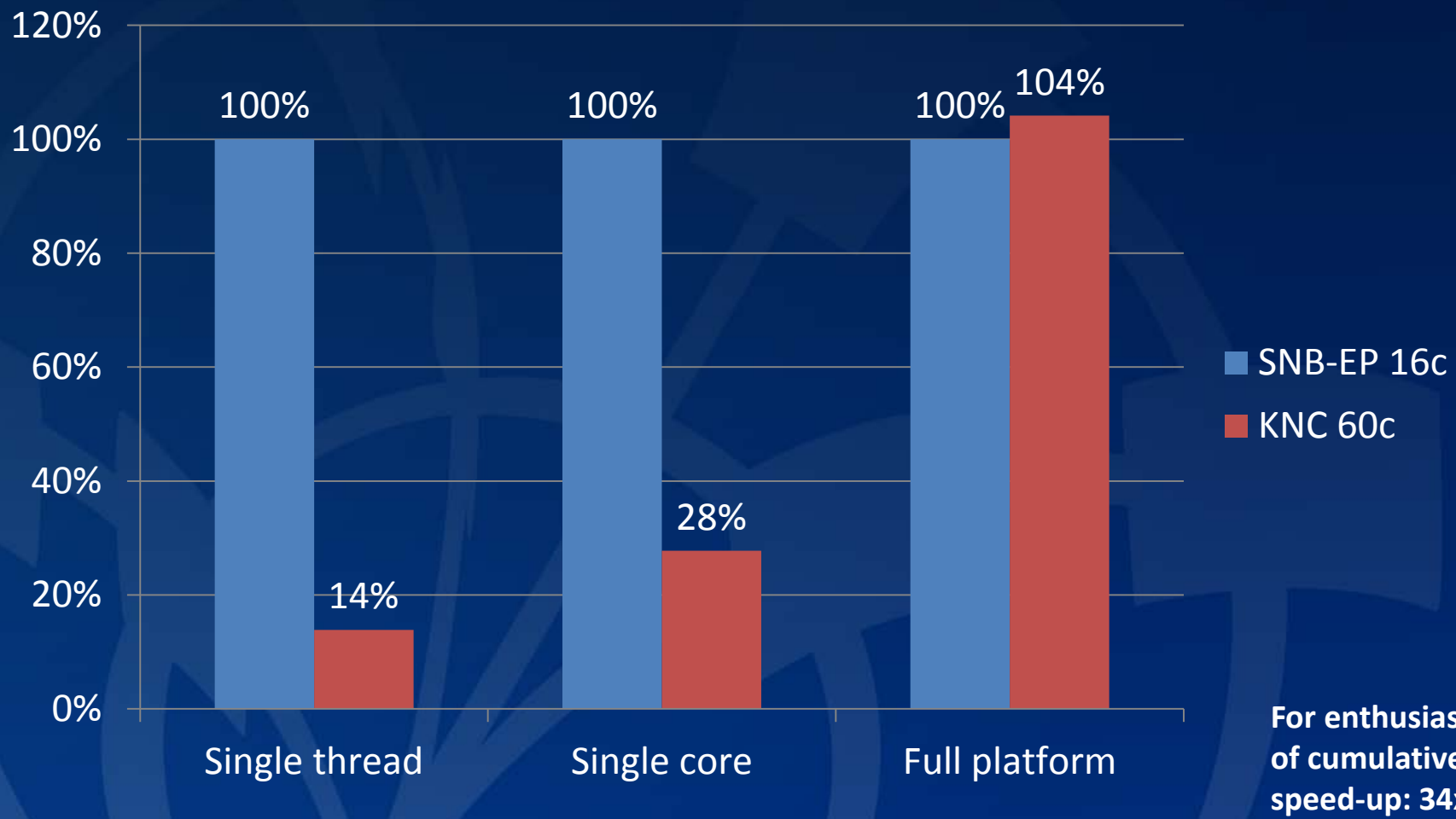
Track fitter throughput
(higher is better)



For enthusiasts of cumulative speed-up: >100x

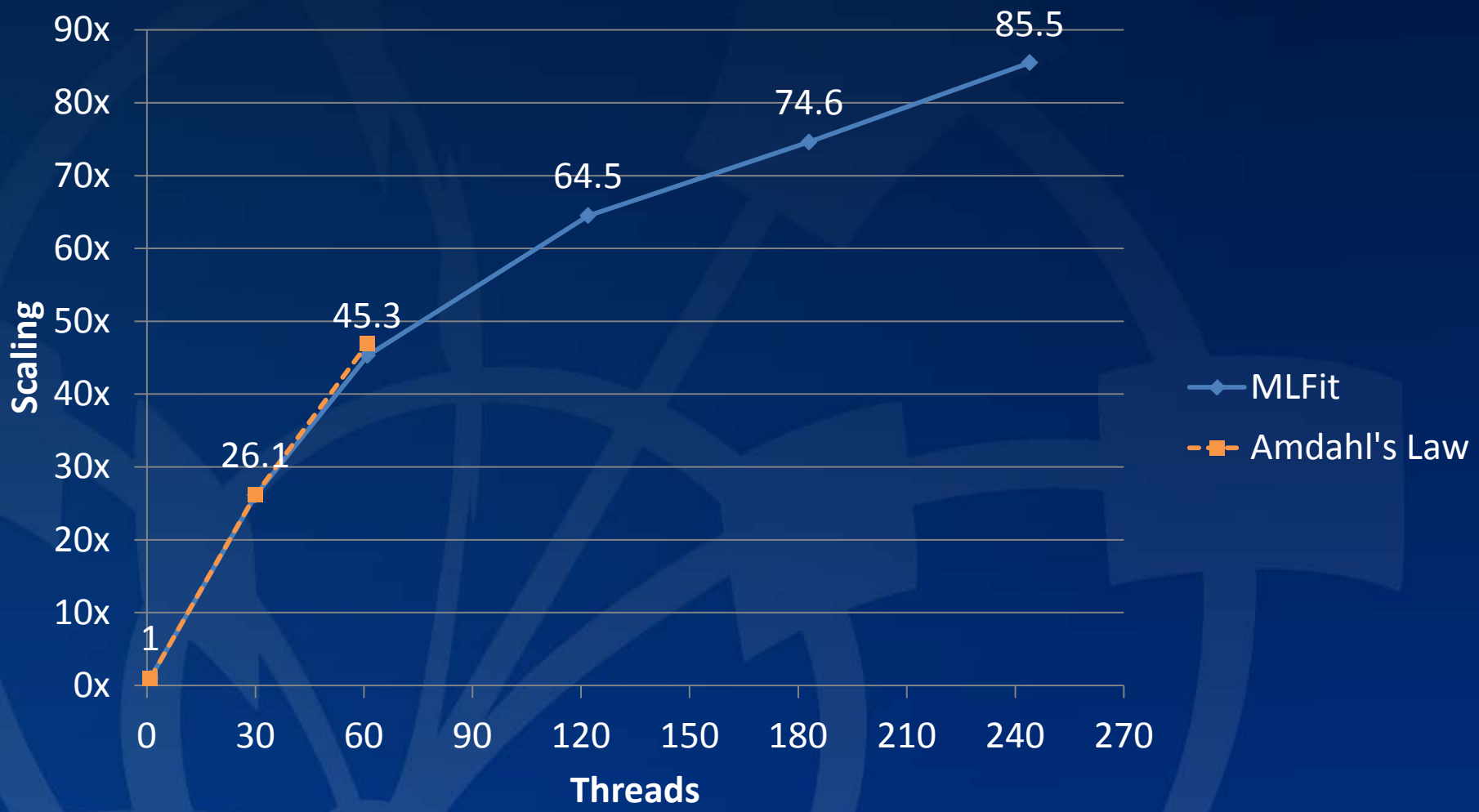
MLFit performance

OpenMP, no block splitting, higher is better

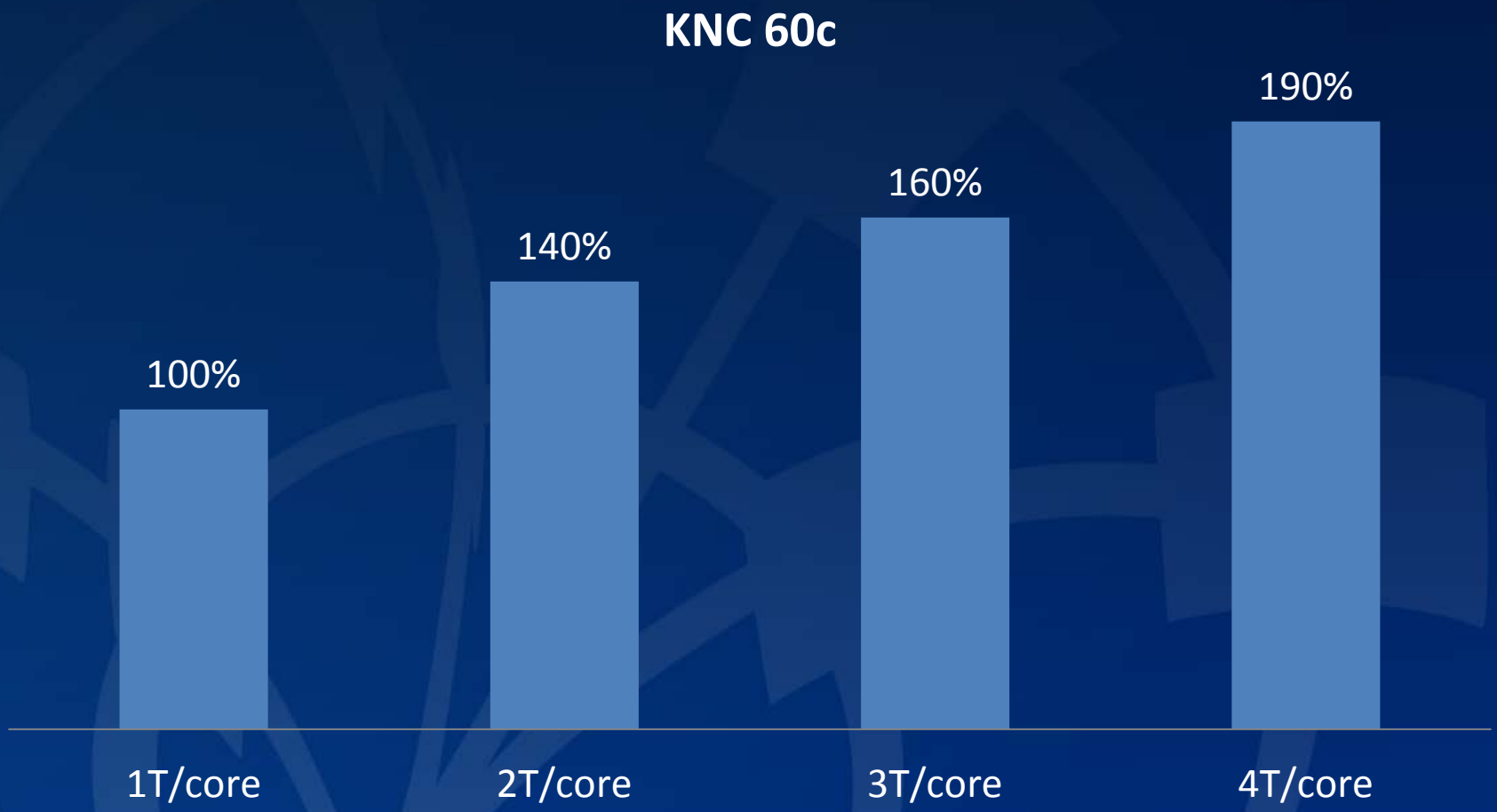


For enthusiasts of cumulative speed-up: 34x

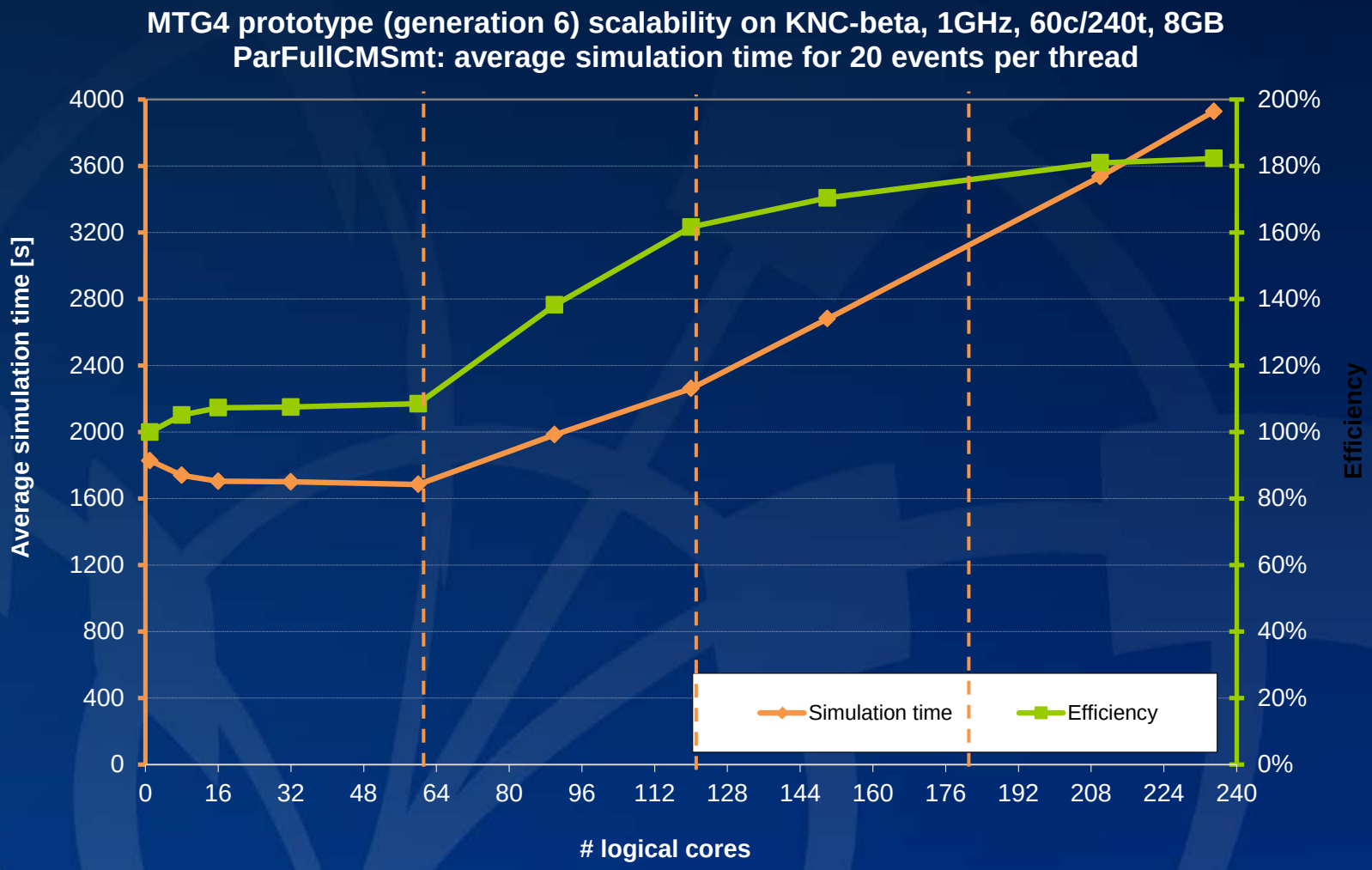
MLFit scaling
(OpenMP)



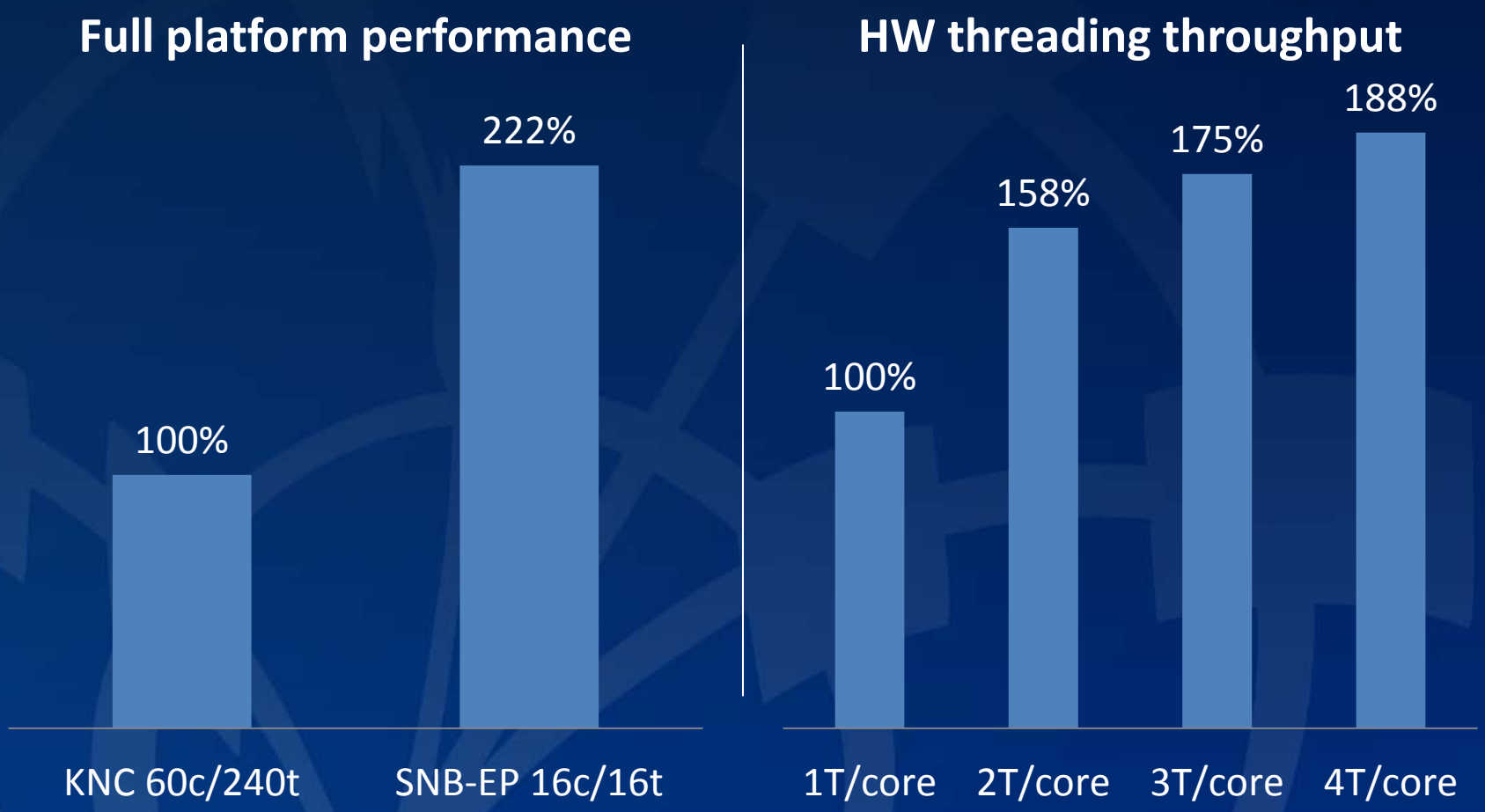
MLFit threading performance



MTG4 scalability



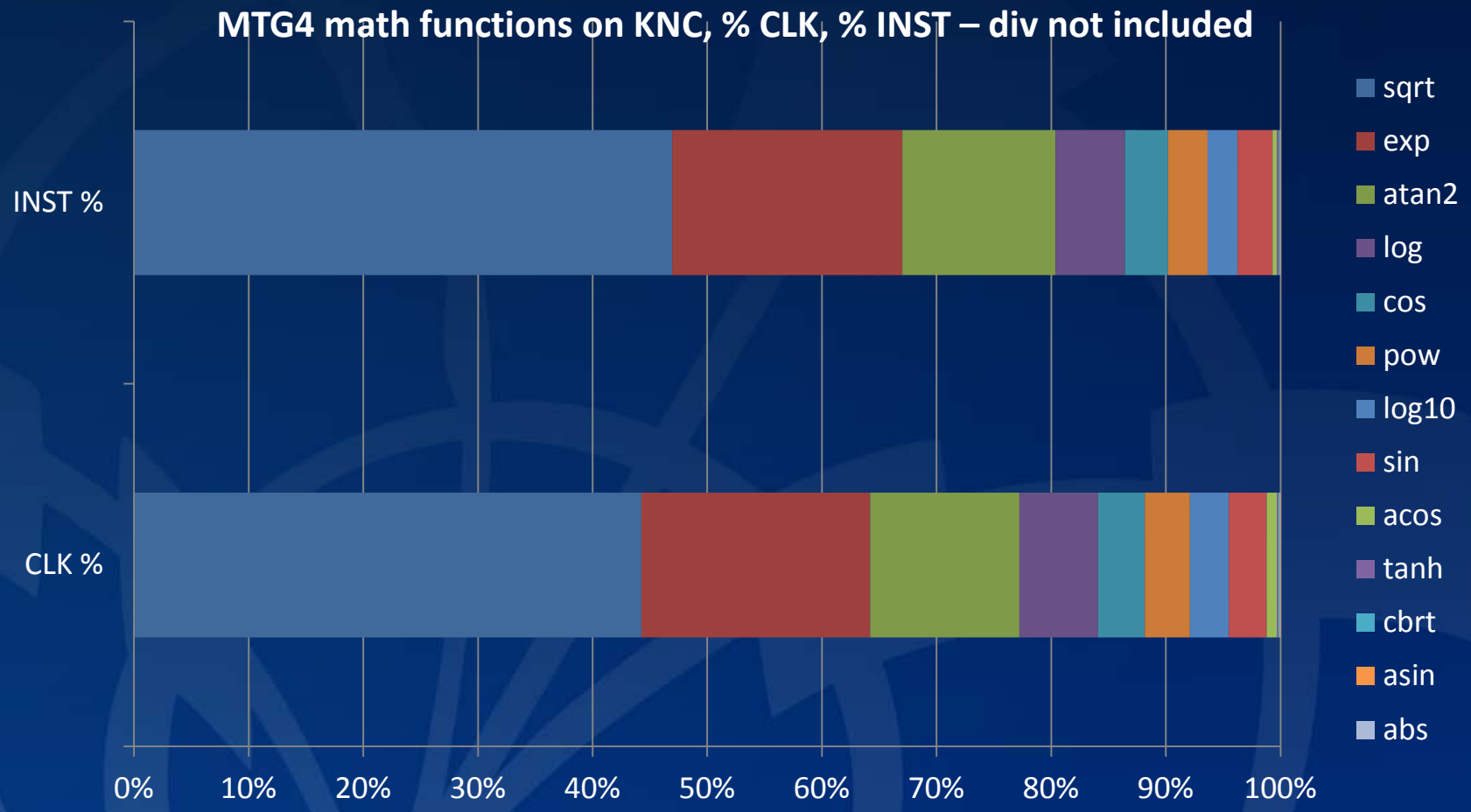
MTG4 performance
(higher is better, no vectorization)



MTG4 – example profile

Function / Call Stack	CLK %	INST %
sqrt	14.35%	22.16%
exp	6.47%	9.47%
atan2	4.22%	6.31%
CLHEP::RanluxEngine::flat	3.24%	5.60%
G4ElasticHadrNucleusHE::HadronNucleusQ2_2	3.01%	2.41%
G4PhysicsVector::Value	2.76%	0.95%
log	2.22%	2.85%
G4VoxelNavigation::LevelLocate	2.05%	0.66%
G4VoxelNavigation::ComputeStep	1.64%	1.10%
G4ClassicalRK4::DumbStepper	1.59%	2.96%
G4SteppingManager::DefinePhysicalStepLength	1.54%	1.39%
G4Navigator::ComputeStep	1.40%	1.01%

MTG4 math


MTG4 math functions on KNC, % CLK, % INST – div not included
INST %
CLK %
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
sqrt
exp
atan2
log
cos
pow
log10
sin
acos
tanh
cbrt
asin
abs

Performance – observations

- Optimized applications surpass dual-socket Xeon performance
- Vectorization is key to success
- Non-optimized performance reaches approximately a single Xeon socket
- Math function usage and performance are key vis a vis Xeon
- Compiler maturity still an open question
  - Some improvements already in the pipeline

Multiple dimensions of performance

Dimension	Software
Nodes	MPI
Sockets	Threading and NUMA control
Cores	Threading
ILP	Efficient compilers and code
Pipelining	Efficient compilers
Vectors	Various options

MIC software - scenarios

Native mode

workload runs entirely on a MIC system (networked via PCIe)

Offload

MIC as an accelerator where host gets weak

Balanced

MIC and host work together

Cluster

application distributed across multiple MIC cards (possibly including host)

Where are our benchmarks now?

- Based on our findings, new MIC-enabled Geant4 versions are being produced by the Geant4 team
- MIC-enabled ROOT versions supplied to core ROOT developers
- Trackfitter and MLFit will remain as PoC
- Workload visualizer produced in Summer 2012, designed to work on MIC as well

MIC evolution – implications for HEP

- Small core evolution
- Hybrid mixes (Xeon + MIC)
- ISA convergence
- Will I/O latency or BW be a constraint?
- Other applications:
  - HPC-like code (e.g. QCD, CFD)
  - Triggering
  - High data throughput (ICE-DIP)

MIC evolution: ICE-DIP

- **FP7 project looking for (amongst other things) efficient methods of accelerator/co-processor use**
- **Focus on data taking past LS1**
- **Of particular interest**
  - Getting data into the platform
  - Getting data into the accelerator/co-processor
  - Efficient processing
  - Efficient distribution of results

Practical observations

- **The physics community is keen, but cautious**
  - A young product
  - Lack of OSS tools
  - Opaque ecosystem
  - Porting is easy, but getting performance on existing code requires more work than anticipated
- **Competition from GPUs**
  - Price
  - Marketing
  - Vendor responsiveness

How can HEP benefit?

- **Pros:**
  - Ultimate programmability
  - Porting is a breeze (if porting at all)
  - Wide vectors, many threads
  - Cheap communication
  - Promise of good performance, especially for HPC-like workloads
  - Multiple cards in a system supported
  - Xeon optimization payoff
- **Cons:**
  - Host attachment currently only PCIe
  - Limited amount of on board memory
  - Latency guarantees unknown
- **Performance on highly optimized workloads is 2-3x better than obtained on a dual-socket Xeon**

Are you interested?

- **We are always looking for interesting prototypes from the physics community**
- **We recommend:**
  - Data oriented design as opposed to only control flow oriented (OOO) – no vectorization, no fun
  - Porting to the Intel compiler on Xeon first
  - Thinking in multiple dimensions of performance – vectorize, thread, limit memory usage
  - Checking precision and math usage

THANK YOU

Q & A

Questions? Andrzej.Nowak@cern.ch

BACKUP

- Sverre's Minor Review presentation on KNC: http://openlab.web.cern.ch/publications/presentations/vector-architecture-intel-many-core-co-processor-knights-corner