INTRODUCTION TO PERFORMANCE ANALYSIS

Andres S. CHARIF-RUBIAL

William Jalby

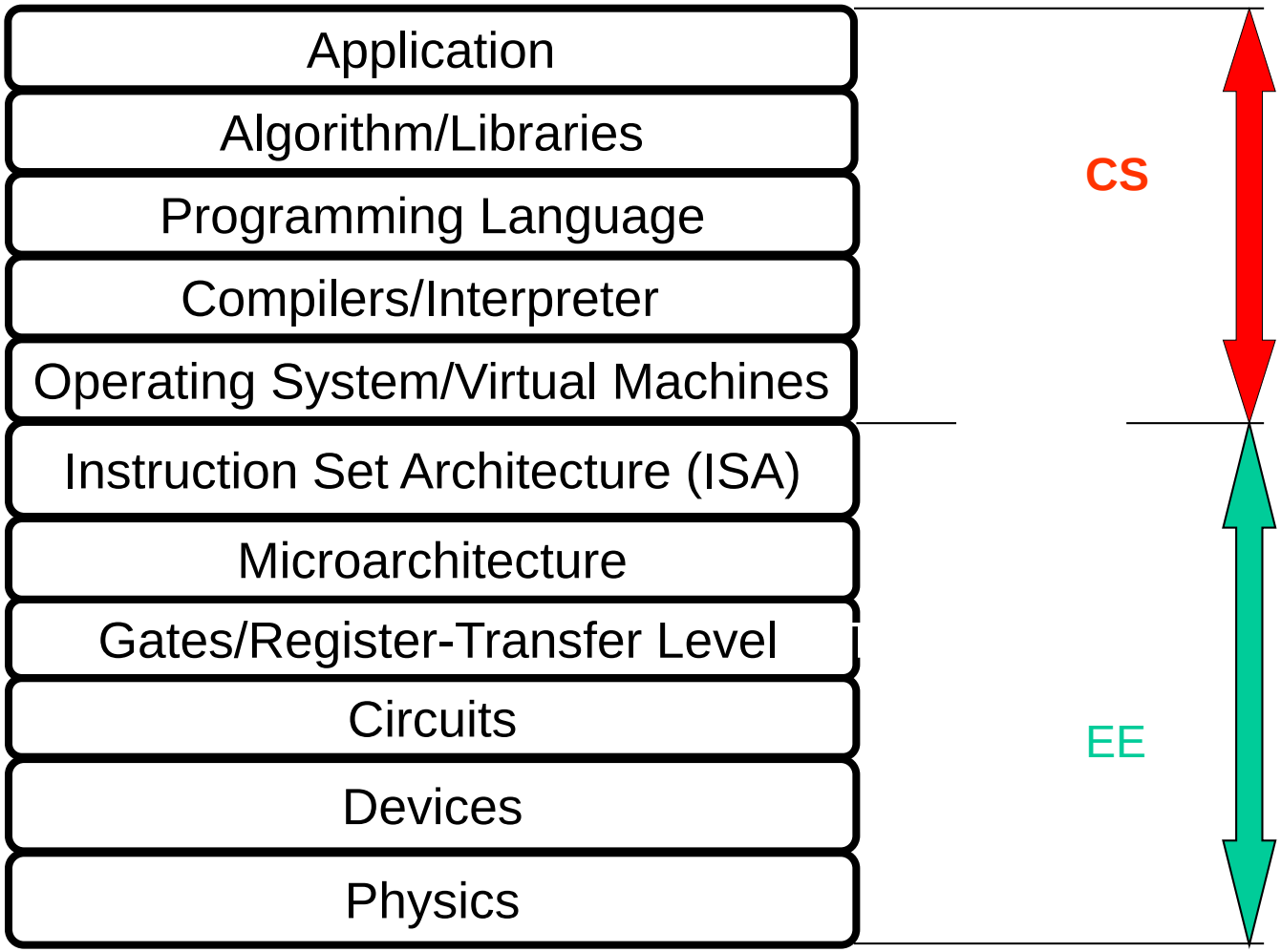
UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

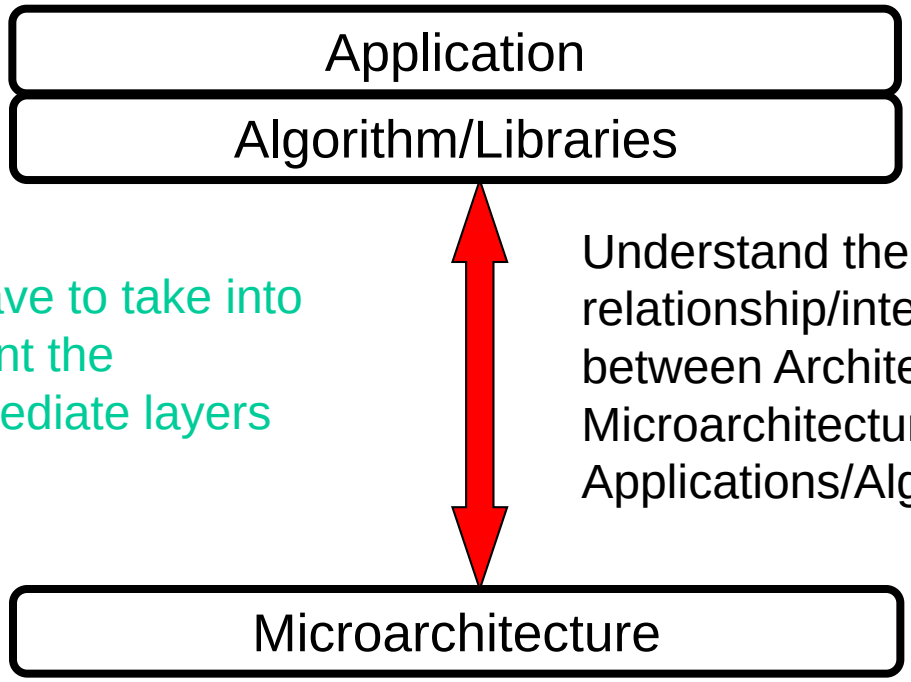
Overview

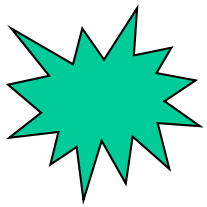
1. The stage/actors/trends
2. Measurement Techniques
3. Limitations of existing tools
4. A case study

Abstraction Layers in Modern Systems



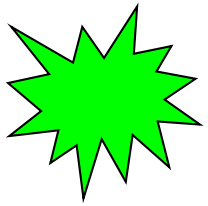
OUR OBJECTIVE/POSITIONNING





We have to take into account the intermediate layers

Understand the relationship/interaction between Architecture Microarchitecture and Applications/Algorithms



Don’t forget also the lowest layers

KEY TECHNOLOGIES:

- Performance Measurement and Analysis

- Compilers

Recent Trends in Computer architecture

- More complex cores: FMA (Fused Multiply Add), wider and more complex vector instructions
- More complex memory hierarchies: multiple levels, multiple hardware prefetch mechanisms,….
- Increase in parallelism: Many core, GPU,

INTEL Processors Roadmap

Tick Tock model

Tick = shrink of an existing micro architecture

Tock = new micro architecture using existing IC process

Nehalem	new micro arch 45 nm	Tera 100
Westmere	new process 32 nm	
Sandy Bridge	new micro arch 32 nm	Curie
Ivy Bridge	new process 22 nm	
Haswell	new micro arch 22 nm	Tera 1000 ??
Broadwell	new process 14 nm	

Haswell

~140W

≥ 440 Gflops

AVX2 (FMA + gather)

22nm

Vector Width: Evolution

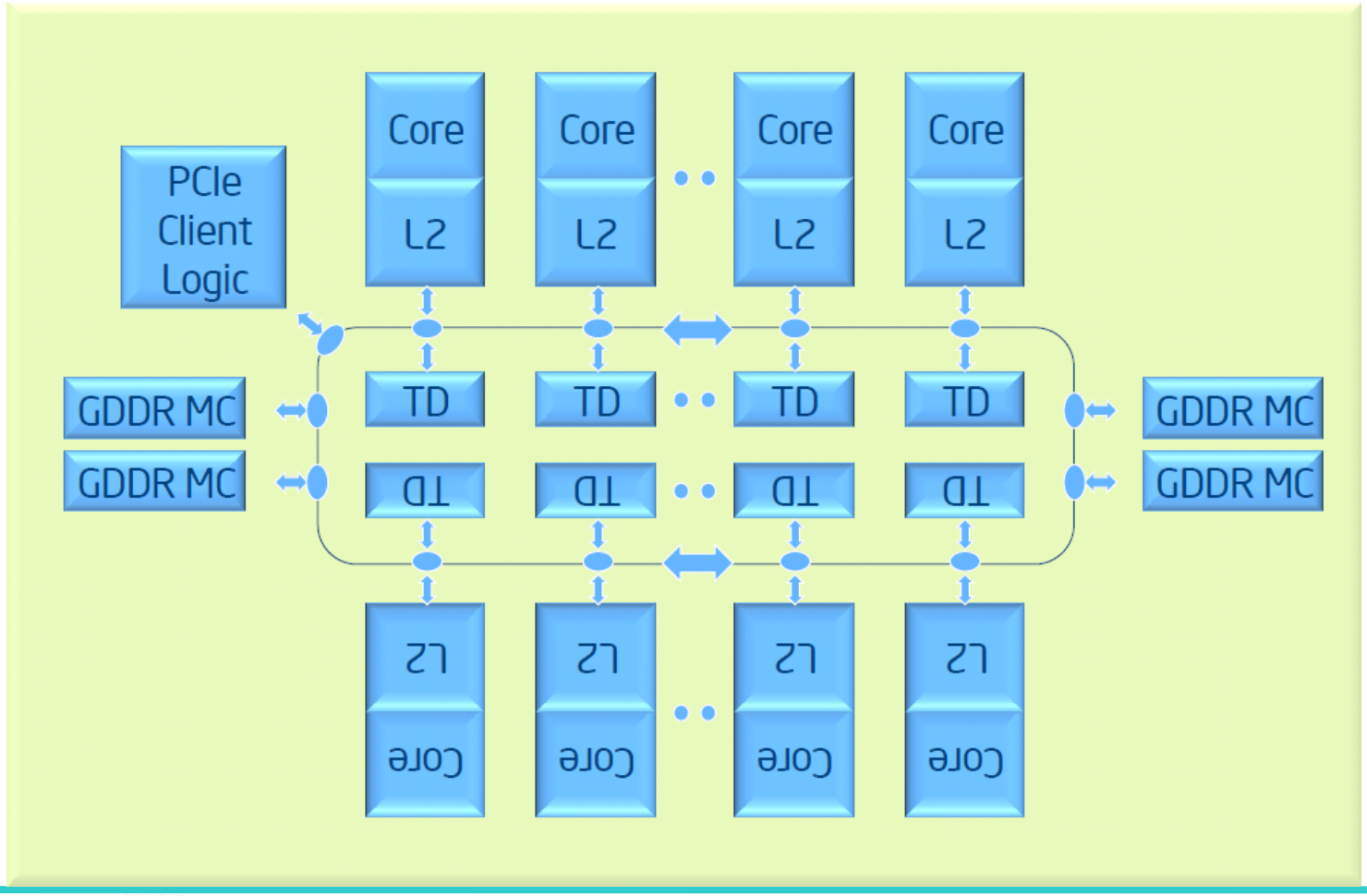
Année	Registres	Nom
~1997	80-bit	MMX
~1999	128-bit	SSE1
~2001	128-bit	SSE2
~2010	256-bit	AVX
~2012	512-bit	ABRni (KNC)
~2014	256-bit	AVX2 (Haswell)

Register Organization: SSE/AVX/MIC

<table>
<tr><td colspan="3">511</td><td colspan="4">255</td><td colspan="4">127</td><td colspan="4">0</td><td></td><td></td></tr>
<tr><td colspan="8"></td><td colspan="4">DP 1</td><td colspan="4">DP 0</td><td>2DP FP</td><td>SSE</td></tr>
<tr><td colspan="8"></td><td>SP 3</td><td>SP2</td><td>SP 1</td><td>SP 0</td><td>4SP FP</td><td></td></tr>
<tr><td colspan="3"></td><td colspan="4">DP3</td><td colspan="2">DP2</td><td colspan="4">DP 1</td><td colspan="4">DP 0</td><td>4DP FP</td><td>AVX 256</td></tr>
<tr><td colspan="3"></td><td>SP 7</td><td>SP 6</td><td>SP 5</td><td>SP 4</td><td>SP 3</td><td>SP 2</td><td>SP 1</td><td>SP 0</td><td>8SP FP</td><td></td></tr>
<tr><td colspan="2">DP 7</td><td>…</td><td colspan="4">DP 3</td><td colspan="2">DP 2</td><td colspan="4">DP 1</td><td colspan="4">DP 0</td><td>8DP FP</td><td>MIC 512</td></tr>
<tr><td>SP 15</td><td>SP 14</td><td>…</td><td>SP 7</td><td>SP 6</td><td>SP 5</td><td>SP 4</td><td>SP 3</td><td>SP 2</td><td>SP 1</td><td>SP 0</td><td>16SP FP</td><td></td></tr>
</table>

Xeon Phi Architecture (KNC)

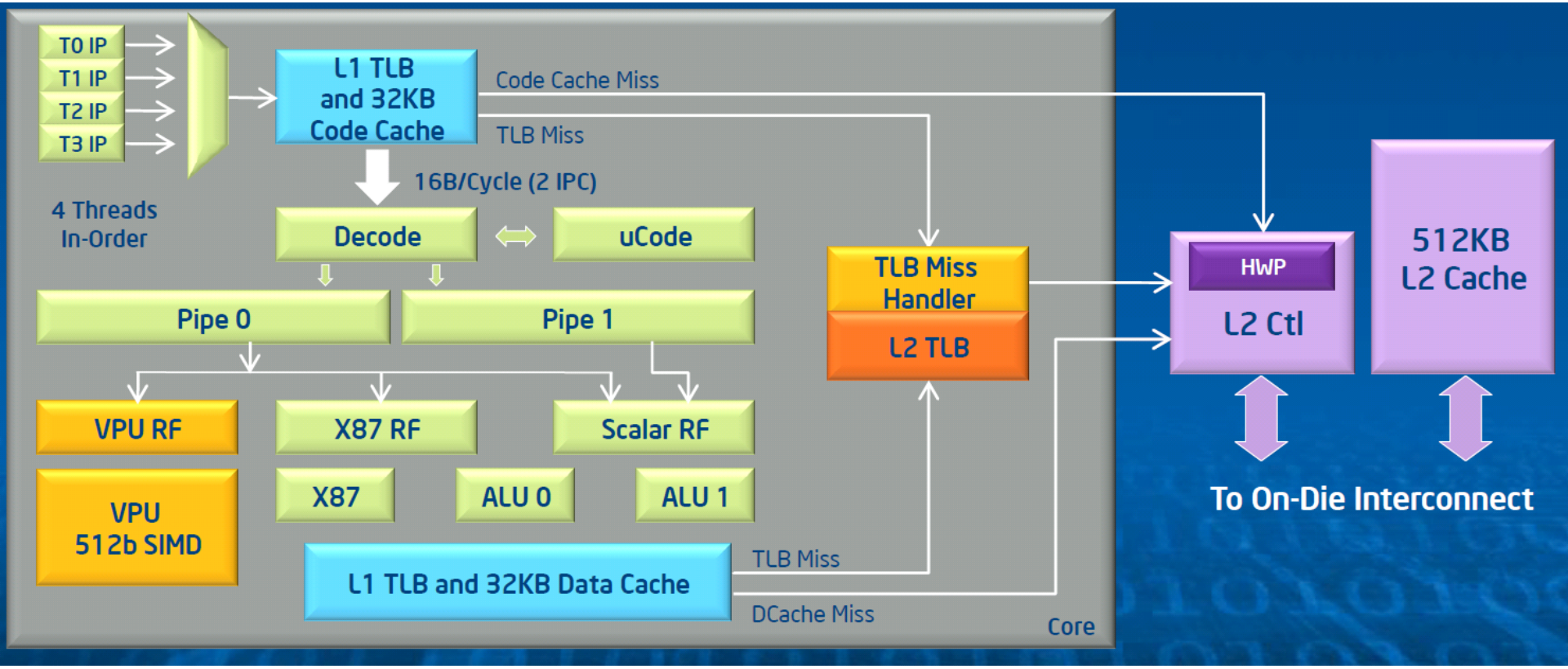
61 cores, 4 threads per core



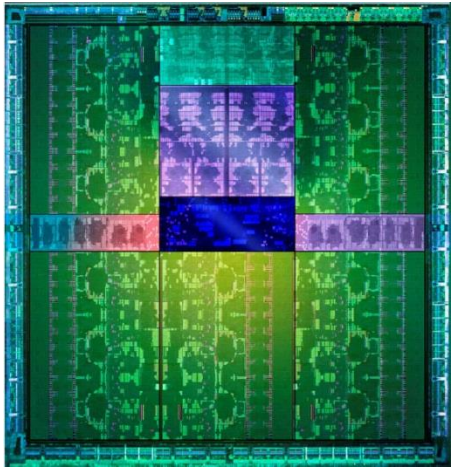
Xeon Phi Core: detailed view

Architecture in order (old P45 cf. Pentium Pro).

Next generations will be « out of order » / on socket



The GPU path



NVIDIA Kepler 2

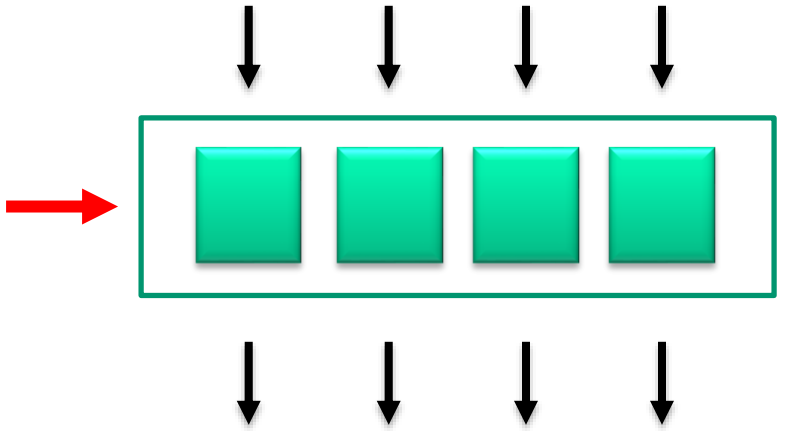
- 960DP + 2880 SP cores ~1.5TFlops DP
- 7.1 B transistors
- <300W

Standard goals for Performance Analysis

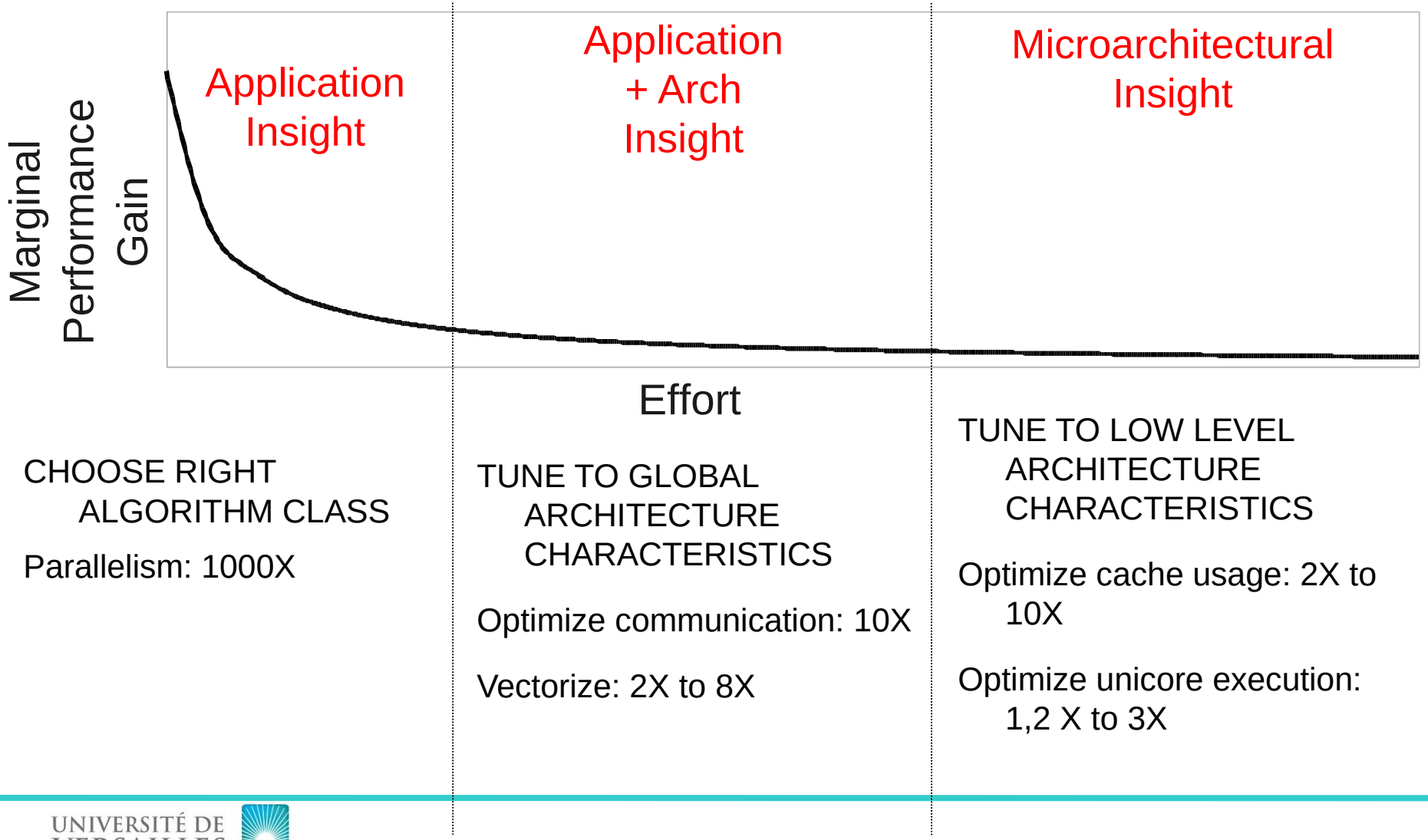
- For a given architecture and application, improve application performance: tune performance and/or change algorithms.
- For a given set of applications, try to determine best architecture including its variants (cache size, memory/core organization etc …)
- For Computing Center managers, optimize resource usage
- For hardware/system designers, understand bottlenecks on current architectures and derive guidelines for next generation
- NEW: For a given architecture and application, improve its energy consumption

Analysis levels

- Inter-Node
- Node
  - Sockets
- Core level
  - SIMD: data //
  - ILP: instruction level //
  - Remote memory
  - Remote cache



Performance tuning curve



CHOOSE RIGHT ALGORITHM CLASS

Parallelism: 1000X

TUNE TO GLOBAL ARCHITECTURE CHARACTERISTICS

Optimize communication: 10X

Vectorize: 2X to 8X

TUNE TO LOW LEVEL ARCHITECTURE CHARACTERISTICS

Optimize cache usage: 2X to 10X

Optimize unicore execution: 1,2 X to 3X

Performance Tuning

- Generally a multifaceted problem
  - Maximizing the number of views
- Identify clearly performance issues:
  - Where ?? source code fragment (ideally a few statements)
  - Who ?? algorithm, compiler, OS, hardware
  - How much ?? exact cost of performance issues
- Three solution techniques
  - Analytical models
  - Simulation
  - Measurements

Analytical Models

Mathematical equations describing system (or more likely subsystem) performance in function of key parameters

- ✚ Allows to exactly capture impact of parameters and ideal for performance tuning
- ✚ Fast
- ▬ Requires very strong simplifying assumptions to remain tractable/usable: low accuracy
- ▬ Has to be validated/calibrated against simulation/experiment

- Exemples
  - ➢ Amdahl's law (estimate performance gain)

Simulation

Software tool modeling hardware behavior of system or subsystem

- ✚ Explicit direct relation between hardware and software
- ▬ Slow: accuracy versus speed trade off (OS impact often not taken into account)
- ▬ Has to validated/calibrated against experiment
- ▬ To be accurate requires deep knowledge on target architecture

- Examples
  - ➢ Cache simulators: good tool to apprehend program temporal locality

Measurements

Direct measurement of running programs

- ✚ Excellent accuracy (if measurements done correctly): everything taken into account, no simplifying assumption: IDEAL
- ✚ Fast (not so fast if good measurement methodology is used)
- ▬ Difficult to vary parameters
- ▬ Difficult separate parameters impact (aggregate effect)

- Examples
  - ➢ Analytical models built using measurement (microbenchmarks)

Metrics

- What can be measured:
  - Counts of a given hardware event occurrences: cache miss, instruction stalls, etc …
  - Time: time interval
  - Values: value profiling: stride of memory access, loop length, message size etc ….
- Difficulties:
  - Accuracy
  - Correlation with source code: aggregate values (total number of cache misses for the whole loop not for individual statements)

TIME

- Wall clock time: it includes everything: I/O, system etc ….. Including other programs running simultaneously but it corresponds to response time
- CPU Time:
  - Time spent by CPU to execute programs
  - Real target
- How to measure time ?? recommendation use RDTSC: Read Time Stamp Counter (assembly instruction with good accuracy). However small durations (less than 100 cycles are extremely difficult to measure if not impossible)

Derived Metrics

- Rates: obtained by dividing number of occurrences by time
  - GIPS Billions of Instructions per second
  - GFLOPS Billions of Floating point instructions per second
  - MBYTE/s number of Mbytes per second (useful for characterizing stress on various memory levels)
  - THROUGHPUT: how many job instances executed per second
- Rates are useful to assess how well some hardware parts are used.
- A useful derived metric: SPEEDUP: T1/Tp Where T1 (resp. Tp execution time on 1 (resp. p) core(s).

How to perform measurements ??

- How to trigger measurements ??
  - Hardware Driven: sampling, counting
  - Code Driven: tracing
- For tracing, how to insert probes ??
  - Level: source, library ,binary
  - Instrumentation: static/dynamic
- Three key questions:
  - How much perturbation is introduced ??
  - How to correlate with source ??
  - How to Record/Display information??

Sampling (1)

- OPERATION MODE (hardware driven):
  1. Focus on a given hardware event: clock ticks, FP operations, cache miss,
  2. At each event occurrence, counter is incremented
  3. When threshold is reached (counter overflow), interrupt occurs and counter reset to 0
- What happens on interrupt ??
  - ➢ Record instruction pointer and charge the whole occurrences count to that IP
  - ➢ Advanced mechanism on INTEL processors: PEBS (Precise Event Based Sampling): record processor state (register values etc …)

Sampling (2)

KEY PRINCIPLE: general statistical measurement techniques relying on the assumption that a subset of the population being monitored is representative of the whole population

- CORRELATION WITH SOURCE CODE:
  - Function level, Basic Block Level, Loop level but NOT AT THE INSTRUCTION LEVEL (reasonably)
  - IP is not enough, whole call stack is needed which is not easy ☺
  - Inclusive Versus Exclusive issue
  - Call site issue

  EXCELLENT EXAMPLE: XE Amplifier (VTUNE/PTU) : INTEL

Inclusive versus Exclusive

Subroutine toto1 (.....)

Basic Block 1 (BB1)

Call toto2

Basic Block 2 (BB2)

Return

Toto2 is leaf in the call graph

INCLUSIVE TIME:

Tinc = T(BB1) + T(toto2) + T(BB2)

EXCLUSIVE TIME

Texc = T(BB1) + T(BB2)

Exclusive time is easy but Inclusive time needs call stack

Issue with call sites

Subroutine toto1

……

call toto2 (4)

…….

call toto2 (10000)

……

Return

Usually, all of the counts relative to the different occurrences of toto2 will be lumped together: bad correlation with source code.

TRICK: use toto2short and toto2long to distinguish the two!!

SAMPLING: pros and cons

PROS

- Binary used as is (no recompile/no modifications)
- User transparent
- Low overhead if sampling period is large
- PEBS offers very interesting opportunities (whole processor state)

CONS

- Accuracy
- Correlation with source code
- Difficult to assert its quality

TRACING

- OPERATION MODE (code driven):
    1. Insert probes (source/library/binary, static/binary) at point of interest (POI)
    2. Measurement performed when probe is executed
    3. Record tracing event/build trace
- Trace formats
    - VTF : Vampir Trace Format
    - OTF1/2: Open Trace Format
    - …

Instrumentation: Probe Insertion

- Source level: e.g. TAU source code instrumenter
- Library level: e.g. PMPI
- Binary level: e.g. MAQAO/MIL
- Probe Insertion
  - Manual: tedious, error prone
  - Automatic: preprocessor, binary rewrite: Might be difficult to select meaningful POI.
  - Automatic by compiler: specification can be done at source level but instrumentation done by compiler: INTEL IFC/ICC 12.0

Source Instrumentation Issue

```
DO I = 1, 200
    DO J = 1, 1000
          ……
    ENDDO
ENDDO
```

Loop Interchange can be performed by compiler

```
DO I = 1, 200
    Start Clock
    DO J = 1, 1000
          ……
    ENDDO
    Stop Clock
ENDDO
```

Loop interchange no longer possible!!

Source Instrumentation: Pros and Cons

PROS

- Portable
- Good correlation with source code

CONS

- Needs recompile
- Interaction with compiler
- Difficult interaction with high level abstractions (C++)
- Requires access to source code

Binary Instrumentation: Pros and Cons

PROS

- No recompile
- Instrument the real target code
- No need to access source code
- Lowest overhead possible
- OK correlation with simple source code constructs.

CONS

- Not portable
- Need access to specialized tooling (disassembler)
- Might be difficult to correlate with High Level abstractions in source code (C++)

Tracing: pros and cons

PROS

- Excellent correlation with source code
- Excellent accuracy
- Traces preserve temporal and spatial relationships between events
- Allows reconstruction of dynamic behavior
- Most general technique

CONS

- Traces can be huge
- How to select POI and events to be measured a priori ??
- Writing large trace files can induce measurement perturbation
- Aggregate view at loop level at best

Context of Performance analysis

Hardware architectures are becoming increasingly complex

Complex CPU: out of order, vector instructions

Complex memory systems: multiple levels including NUMA, prefetch mechanisms

Multicore introduces new specific problems, shared/private caches, contention, coherency

Each of these hardware mechanisms introduce performance improvement but to work properly, they require specific code properties

Performance pathologies: situations potentially inducing performance loss: hardware poor utilization

Individual performance pathologies are numerous but finite

Introduction

(usual performance pathologies)

Pathologies	Issues	Work-around
ADD/MUL balance	ADD/MUL parallel execution (of FMA) underused	Loop fusion, code rewriting e.g. Use distributivity
Non pipelined execution units	Presence of non pipelined instructions: DIV, SQRT	Loop hoisting, rewriting code to use other instructions eg. x86: div and sqrt
Vectorization	Unvectorized loop	Use another compiler, check option driving vectorization, use pragmas to help compiler, manual source rewriting
Complex CFG in innermost loops	Prevents vectorization	Loop hoisting or code specialization

Introduction

(usual performance pathologies)

Pathologies	Issues	Work-around
Unaligned memory access	Presence of vector-unaligned load/store instructions	Data padding, use pragma and/or attributes to force the compiler
Bad spatial locality and/or non stride 1	Loss of bandwidth and cache space	Rearrange data structures or loop interchange
Bad temporal locality	Loss of perf. due to avoidable capacity misses	Loop blocking or data restructuring
4K aliasing	Unneeded serialization of memory accesses	Adding offset during allocation, data padding
Associativity conflict	Loss of performance due to avoidable conflict misses	Loop distribution, rearrange data structures

Introduction

(usual performance pathologies)

Pathologies	Issues	Work-around
False sharing	Loss of BW due to coherence traffic and higher latency access	Data padding or rearrange data structures
Cache leaking	Loss of BW and cache space due to poor physical-virtual mapping	Use bigger pages, blocking
Load unbalance	Loss of parallel perf. due to waiting nodes	Balance work among threads or remove unnecessary lock
Bad affinity	Loss of parallel perf. due to conflict for shared resources	Use numactl to pin threads on physical CPUs

Analysis of current tool set (1)

Lack of global and accurate view: no indication of performance loss (or alternatively ROI)

- Performance pathologies in general but no hint provided on performance impact (cf VTUNE with performance events): we do not know the pay off if a given pathology is corrected
- Worse, the lack of global view can lead you to useless optimization: for example, for a loop nest exhibiting a high miss rate combined with div/sqrt operations, it might be useless to fix the miss rate if the dominating bottleneck is FP operations.
- Source code correlation is not very accurate: for example with VTUNE relying on sampling, some correlation might be exhibited but it is subject to sampling quality and out of order behavior.

Analysis of current tool set (2)

Very often, most of the tools rely on a single technique/approach (simplified view but globally correct)

- Vtune is heavily relying on sampling and hardware events
- Scalasca/vampir is heavily relying on tracing and source code probe insertion
- Sampling aggregates everything together (all instances): might be counterproductive
- In practice, flexibility has to be offered: tracing might be more efficient than sampling and vice versa.

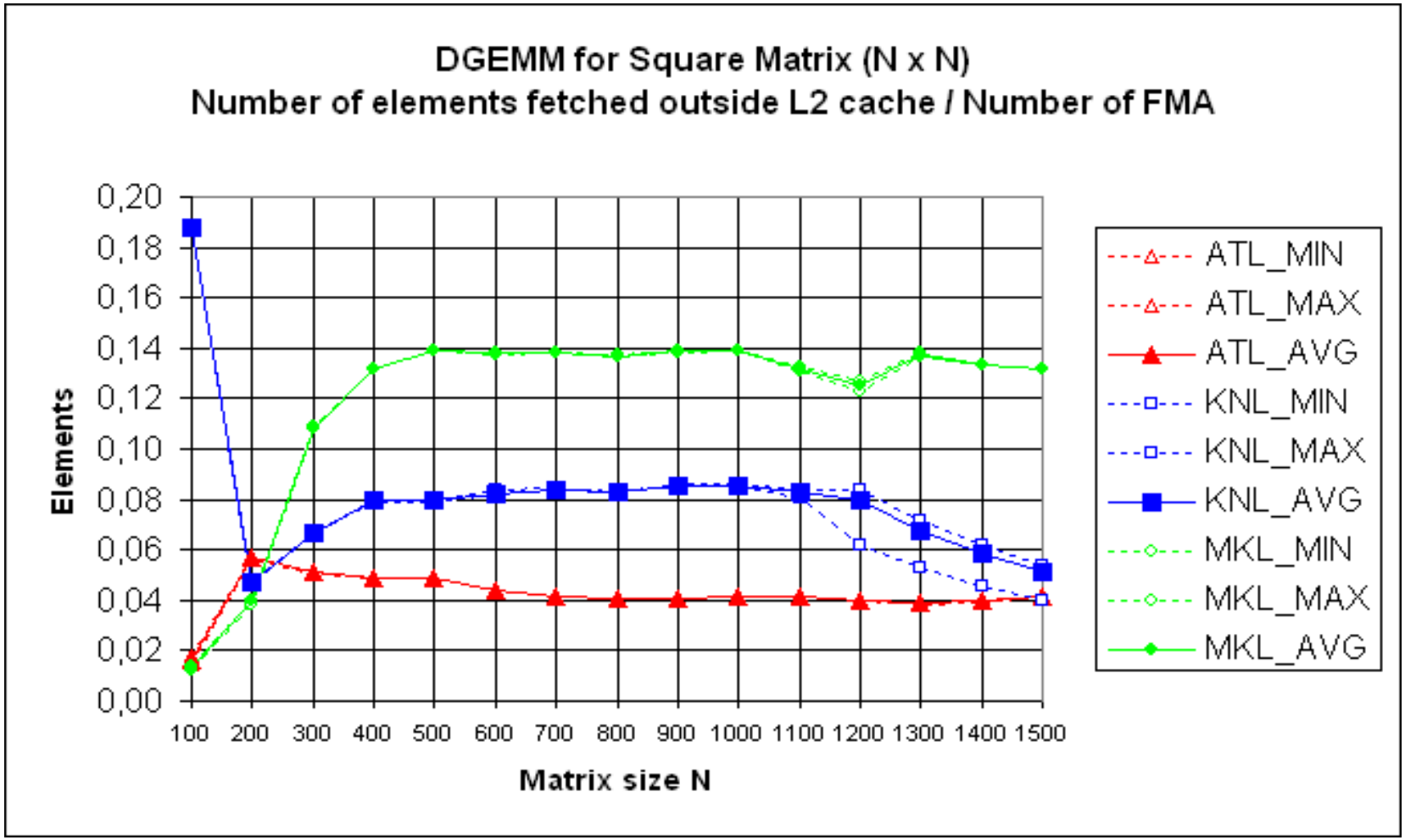
Hardware Performance Counters/Events

- A large number of hardware events (around 1200 on Nehalem processors) can be counted
- BUT DURING A SINGLE RUN, only 4 to 6 counters are available
- Therefore multiple runs are necessary to gather a good set of events
- Multiplexing can increase number of events monitored but at accuracy expense ☺
- Very precise
- Some nice feature: count number of loads exceeding a given latency threshold
- REAL GOAL: hardware debugging. SECONDARY GOAL: understand machine behavior

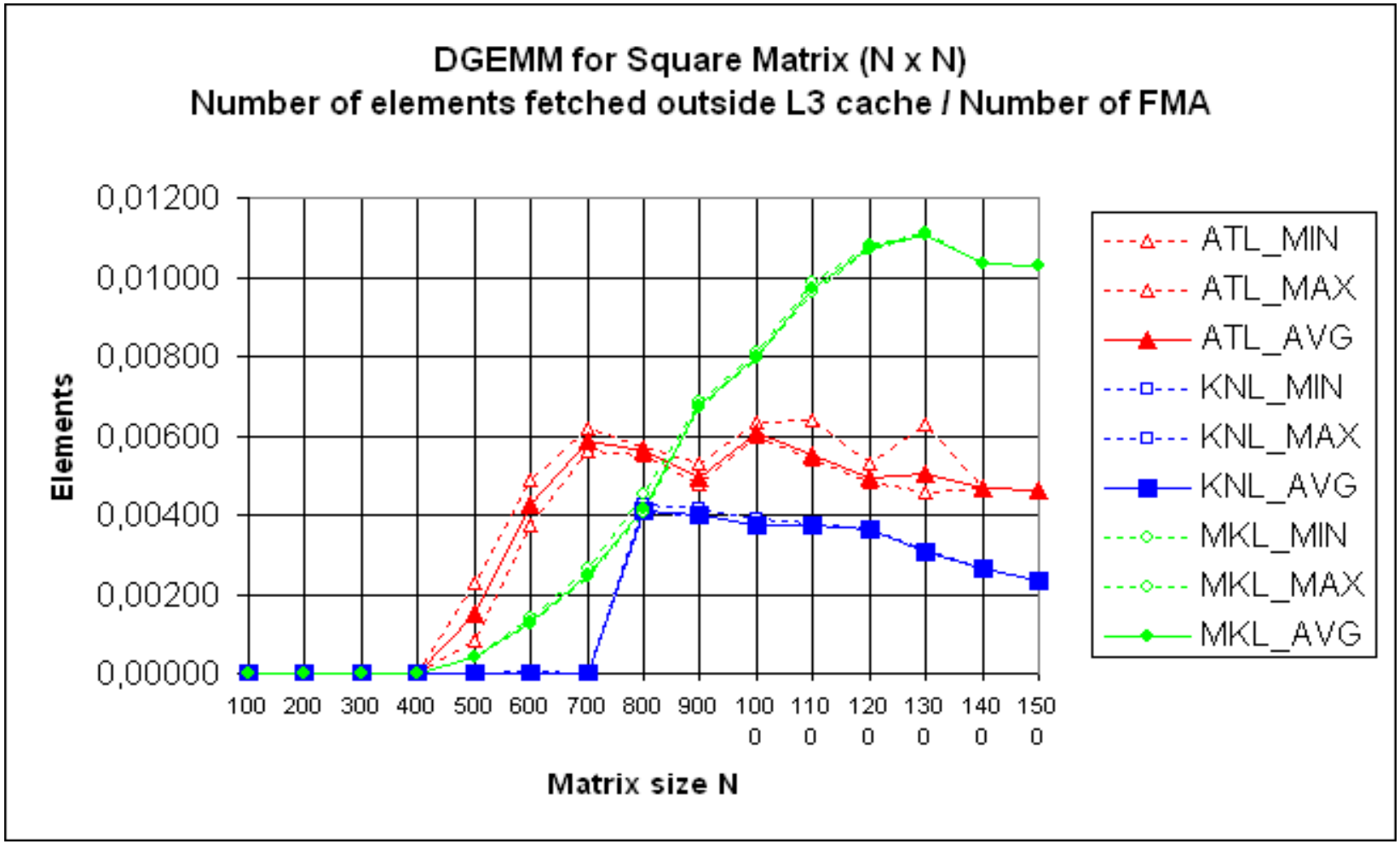
Critics on hardware performance events

- TOO LOW LEVEL: very local view at the hardware level
- NEEDS A DEEP UNDERSTANDING OF MICROARCHITECTURE: no good documentation available on microarchitecture
- CHANGE FROM ONE PROC GENERATION TO THE NEXT: different names designate similar events, same names designate different events
- NEED TO KNOW WHAT TO MONITOR: with 1200 events task is not easy
- HARD TO QUANTIFY: what is high ??
- ALMOST IMPOSSIBLE TO ACCURATELY CORRELATE WITH SOURCE CODE

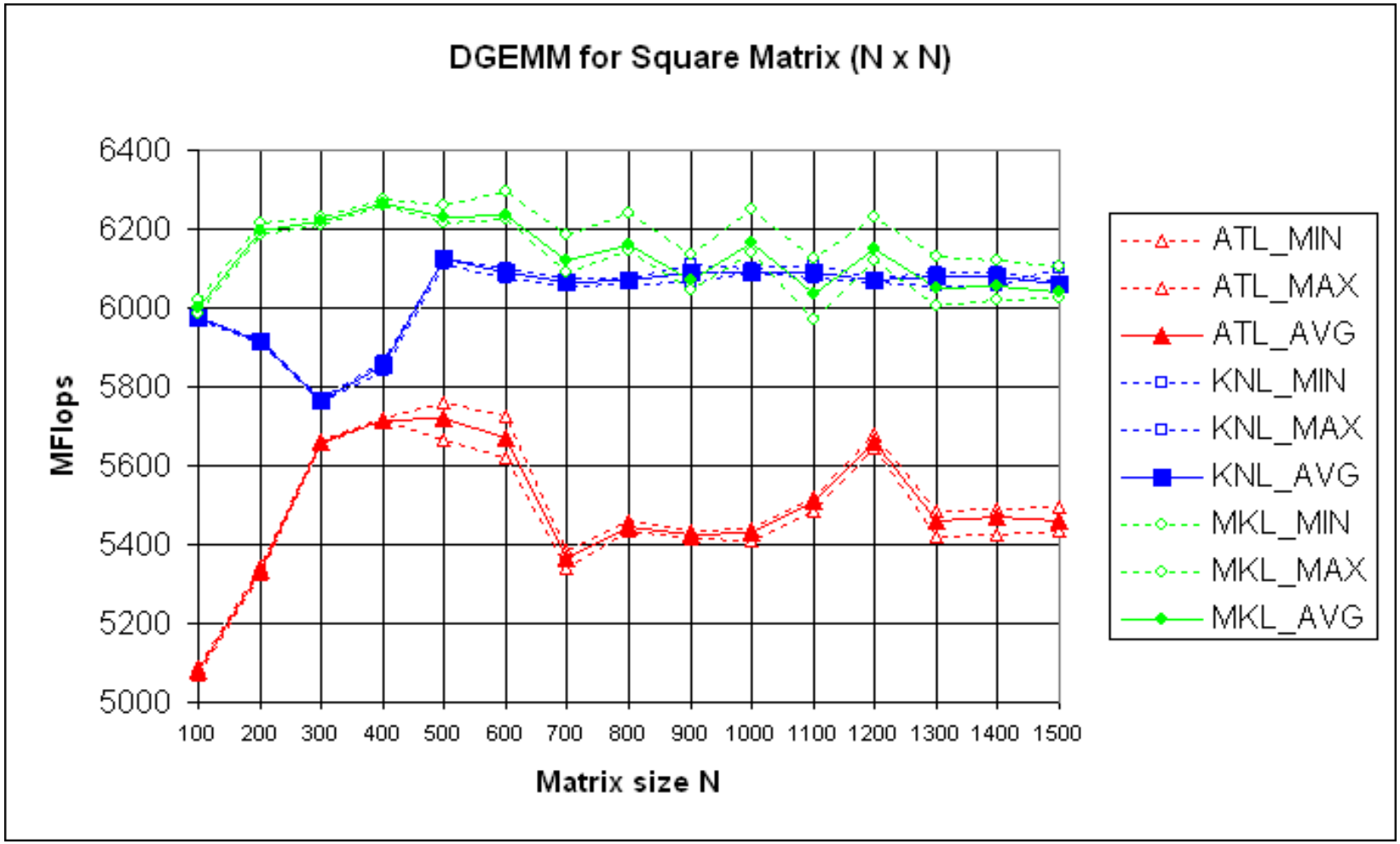
(NxN)(NxN) DGEMM L2 Behavior
Itanium IA 64: ATL = Atlas, KNL= UVSQ optimized



DGEMM (NxN) (NxN) L3 Behavior



(NxN) (NxN) DGEMM Performance



Real Performance Analysis issues

Well known/identified

But:

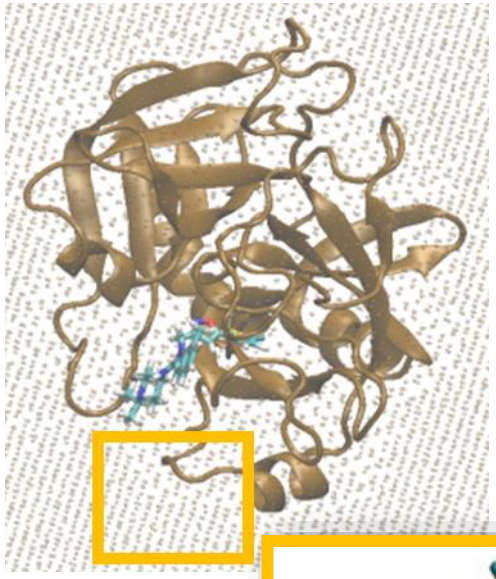
- How to find them ?
- How much do they cost ?
- What to do when multiple pathologies are present ?

Need to quantify/hierarchize them

Case Study (1)
POLARIS(MD) Loop

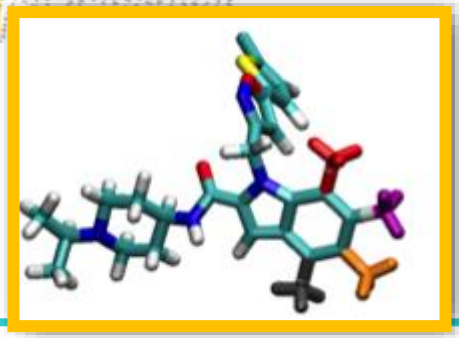
- Molecular Dynamics
  - Based on the Newton equation: $m\vec{a} = -\vec{\nabla}U_{pot}$
  - Multiscale
- Developed at CEA (French energy agency) by Michel Masella
- 60K LOC, Fortran 90
- OMP, MPI, OMP+MPI

Example of multi scale problem: Factor Xa, involved in thrombosis



(7.46 nm)^3

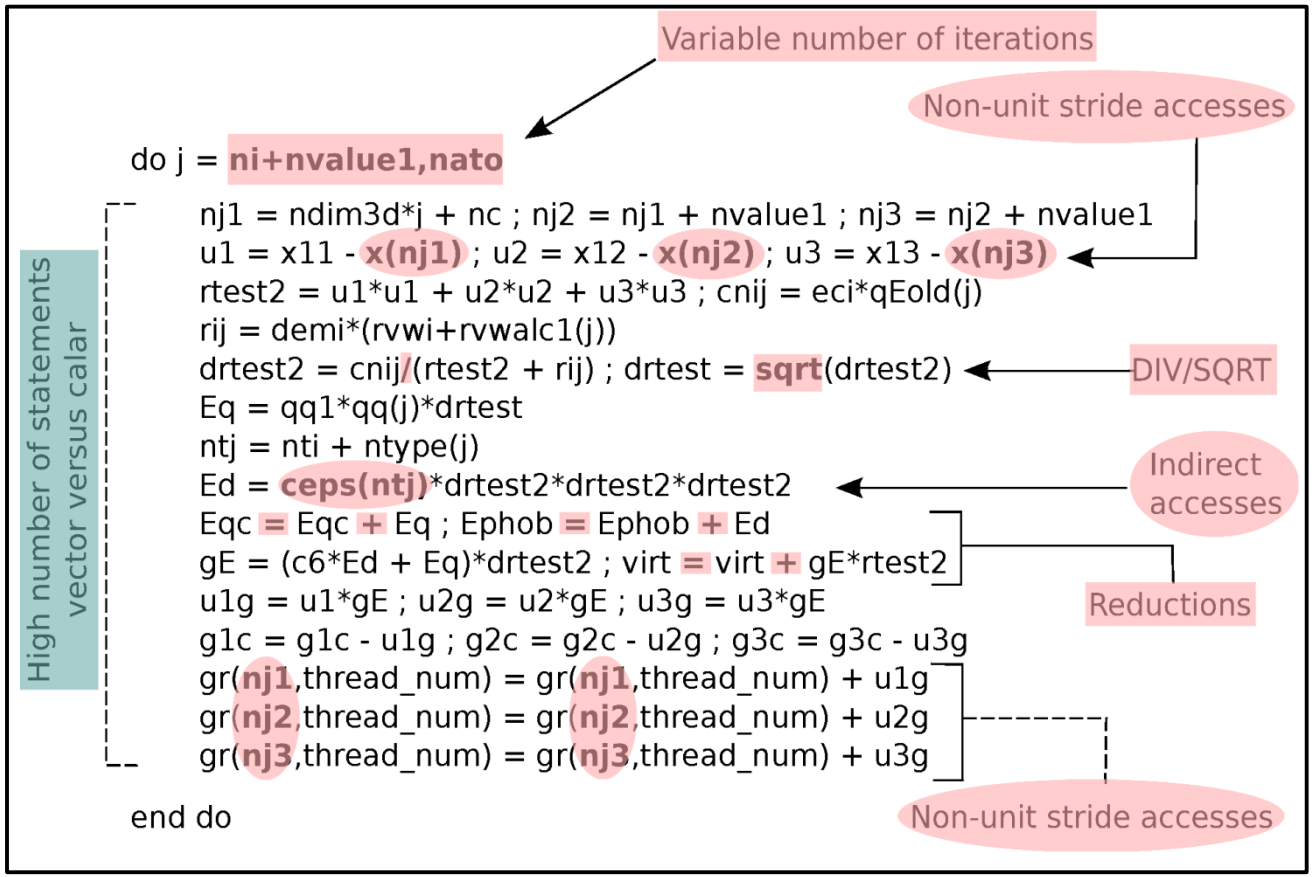
Anti-Coagulant



Case Study (2)

Source code and issues

Loop ~10% walltime



1) High number of statements
2) Non-unit stride accesses
3) Indirect accesses
4) DIV/SQRT
5) Reductions
6) Vector vs scalar

Special issues:
Low trip count: from 2 to 2186 at binary level

Can I detect all these issues with current tools ?

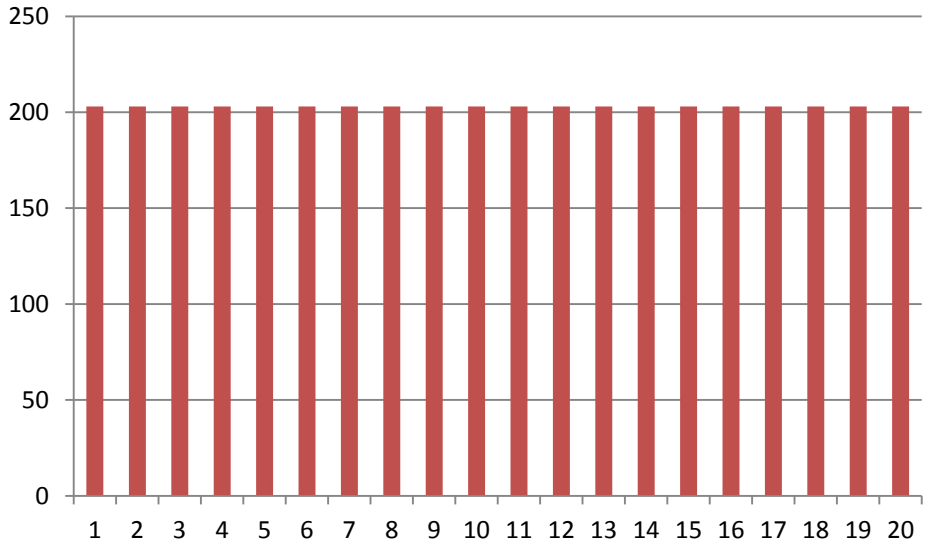
Can I know potential speedup by optimizing them ?

Results obtained using the MAQAO toolsuite and methodology

Case study

Original code : Dynamic properties (1)

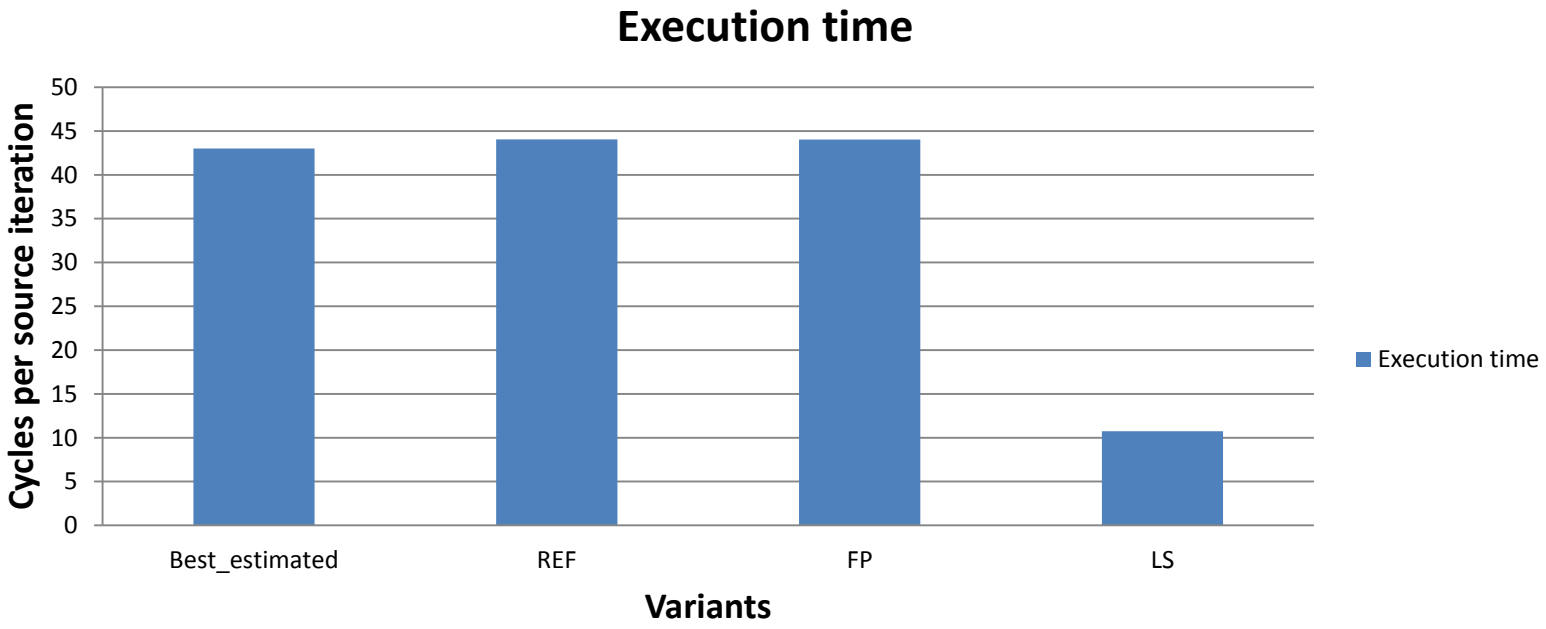
- Trip count: from 1 to 8751 (source iteration count)
- Divide trip count range into 20 equal size interval



All iteration counts are equiprobable (probably triangular access)

Case study

Original code: Dynamic properties (2)



ROI = FP / LS = 4,1

Imbalance between the two streams => Try to consume more elements inside one iteration.

Case study

Original code: Static properties

- Estimated cycles: 43 (FP = 44)
- Vector efficiency ratio: 25% (4 DP elements can fit into a 256 bits vector, only 1 is used)
- DIV/SQRT bound + DP elements:
  - ~4/8x speedup on a 128/256 bits DIV/SQRT unit (2x by vectorization + ~2x by reduced latency)
  - Sandy/Ivy Bridge: still 128 bits
  - => First optimization = VECTORIZATION

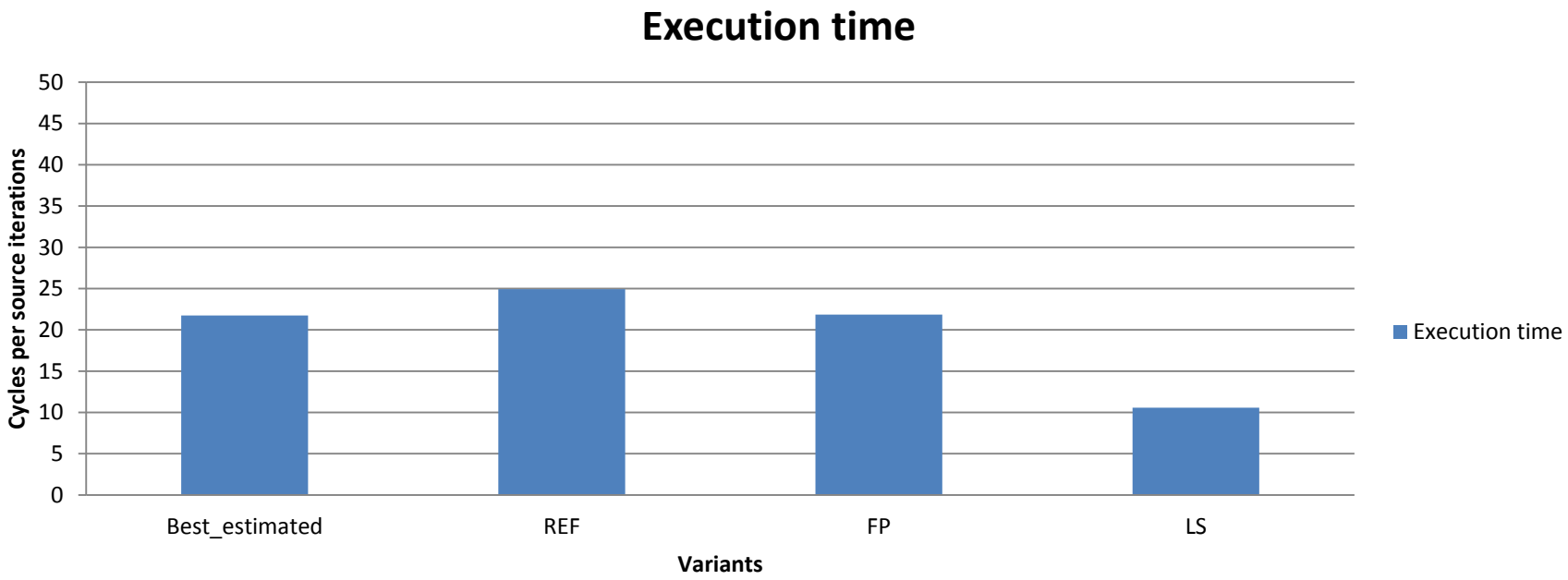
Case study

Vectorization

- Using SIMD directive
- Two binary loops
  - Main (packed instructions, 4 elements per iteration)
  - Tail (scalar instructions, 1 element per iteration)

Case study

Dynamic properties after vectorization

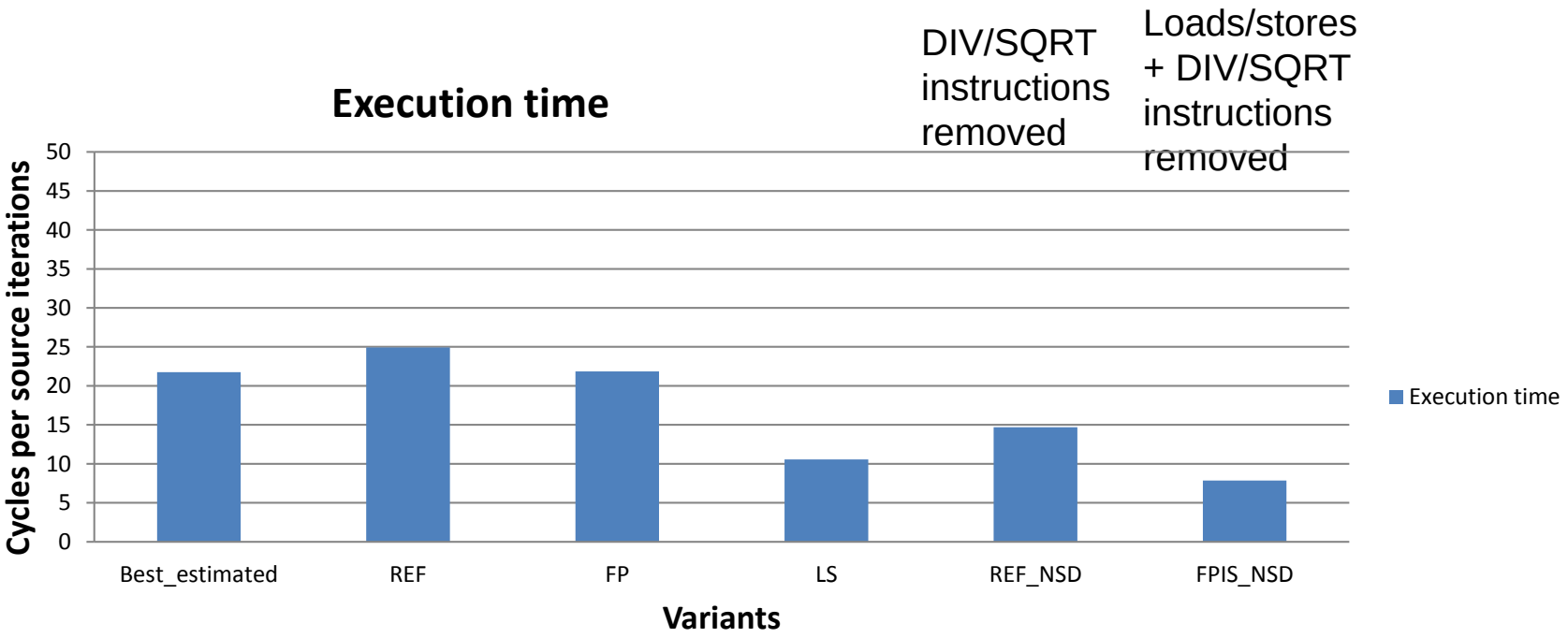


ROI = FP / LS = 2,07 - Initial ROI was at 4,1

removing loads/stores provides a speedup much more smaller than removing arithmetical instructions => focus on them

Case study

Dynamic properties after vectorization



Original_NSD: removing DIV/SQRT instructions provides a 2x speedup
=> the bottleneck is the presence of these DIV/SQRT instructions

FP_NSD: removing loads/stores after DIV/SQRT provides a small additional speedup:
next bottleneck

Conclusion: No space for improvement here (algorithm bound)

Case study

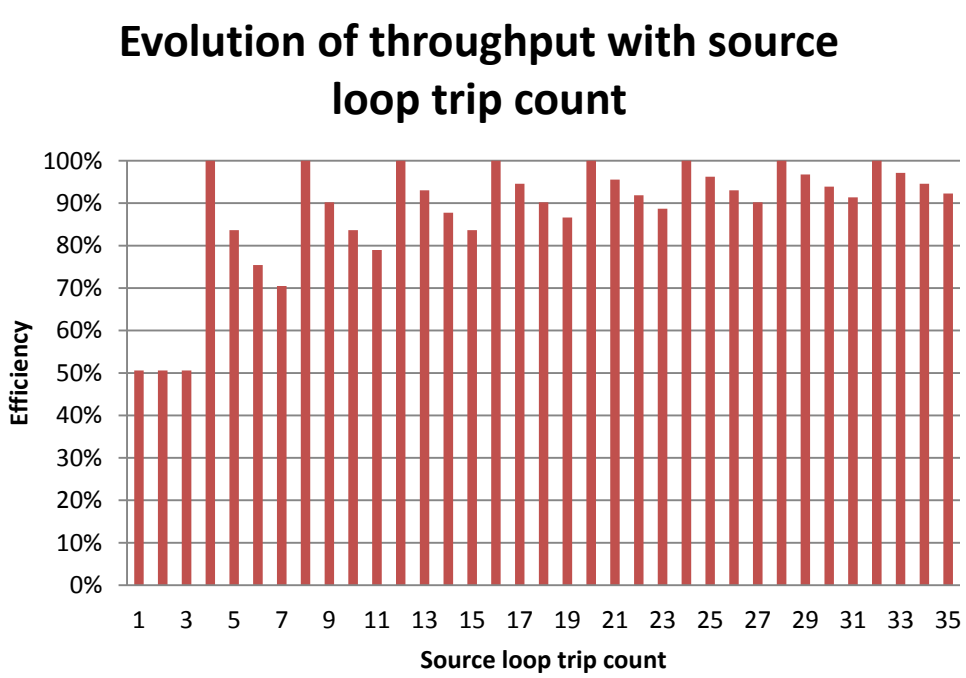
Static properties after vectorization

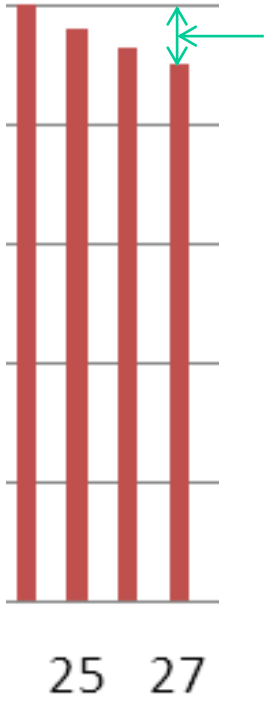
- Vectorization ratio
  - 100% FP arithmetical instructions
  - 65% loads
    - Strided + indirect accesses
    - SCATTER/GATHER not available on Sandy/Ivy Bridge.
- Vector efficiency ratio (vector length usage)
  - 100% FP arithmetical instructions (but 128 bits DIV/SQRT unit)
  - 43% loads (cannot use vector-aligned loads)
  - 25% stores (cannot use vector-aligned stores)

Case study

Static properties after vectorization

- Vectorization overhead: (n/4) x 87 cycles in the main loop vs (n%4) x 43 in the tail loop





With 27 iterations, 10% of time lost due to 3 iterations in the tail loop

Our Objectives

Techniques & Modeling

Get a global hierarchical view of performance pathologies/bottleneck

Estimate the performance impact of a given performance pathology while taking into account all of the other pathologies present

Use different tools for pathology detection and pathology analysis

Perform a hierarchical exploration of bottlenecks: the more precise but expensive tools are only used on a specific well chosen cases

THE 4 KEY ROADBLOCKS

- Algorithm
- Compiler
- OS
- Hardware

Acknowledgements

Material of these slides were produced in part by ECR Performance Evaluation team and UVSQ PerfCloud team (A. Charif Rubial, E. Oseret, Z. Bendifallah, J. Noudohouenou, V. Palomares)

Some slides were borrowed from G. Colin de Verdieres

Thanks for your attention !

Questions ?