Node-Level Performance Engineering for Multicore Systems

J. Treibig

PPoPP 2015, 6.2.2015
The Rules™

There is no alternative to knowing what is going on between your code and the hardware

Without performance modeling, optimizing code is like stumbling in the dark
## Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 – 10:00</td>
<td>Intro / Single-Core Performance</td>
</tr>
<tr>
<td>10:00 – 10:30</td>
<td>Coffee break</td>
</tr>
<tr>
<td>10:30 – 12:00</td>
<td>Node Performance / Performance Tools</td>
</tr>
<tr>
<td>12:00 – 14:00</td>
<td>Lunch</td>
</tr>
<tr>
<td>14:00 – 15:30</td>
<td>Performance Engineering Process</td>
</tr>
<tr>
<td>15:30 – 16:00</td>
<td>Coffee break</td>
</tr>
<tr>
<td>16:00 – 17:30</td>
<td>Performance Modeling / Case Studies</td>
</tr>
</tbody>
</table>
WARMUP:
PERFORMANCE QUIZ
Quiz

- What is a “write-allocate” (a.k.a. read for ownership)?
  - A: Many cache architectures allocate a CL on a store miss.

- What is Amdahl’s Law?

- What is the Roofline Model?


Quiz cont.

- How many cycles does a double-precision ADD/MULT/DIV take?
  A: Intel IvyBridge, ADD 3 cycles, MULT 5 cycles, DIV 21 cycles

- Do you know the STREAM benchmarks?
  A: Defacto standard HPC benchmark for (memory) bandwidth.

- What is SIMD vectorization?
  A: Single instruction multiple data. Data parallel execution units.

- What is ccNUMA?
Where it all started: Stored Program Computer

- Provide improvements for **relevant** software
- What are the **technical** opportunities?
- **Economical** concerns
- Multi-way **special purpose**

**Architect’s view:** Make the common case fast!

EDSAC 1949
Maurice Wilkes, Cambridge
Excursion in memory bandwidth
Some thoughts on efficiency …

Common lore: Efficiency is the fraction of peak performance you reach!

Example: STREAM triad ($A(\cdot) = B(\cdot) + C(\cdot) \ast d$) with data not fitting into cache.

Intel Xeon X5482 (Harpertown 3.2 GHz): 553 Mflops/s (8 cores)
Efficiency 0.54% of peak

Intel Xeon E5-2680 (SandyBridge EP 2.7 GHz) 4357 Mflops/s (16 cores)
Efficiency 1.2% of peak

What can we do about it? Nothing!
Excursion in memory bandwidth
A better way to think about efficiency

Reality: This code is bound by main memory bandwidth.

HPT  6.6 GB/s (8.8 GB/s with WA)
SNB  52.3 GB/s (69.6 GB/s with WA)

In both cases this is near 100% of achievable memory bandwidth.

To think about efficiency you should focus on the utilization of the relevant resource!
Hardware-Software Co-Design? From algorithm to execution

Notions of work:

- Application Work
  - Flops
  - LUPS
  - VUPS

- Processor Work
  - Instructions
  - Data Volume

Algorithm

Programming language

Compiler

Machine code
Example: Threaded vector triad in C

Consider the following code:

```c
#pragma omp parallel private(j)
{
for (int j=0; j<niter; j++) {
 #pragma omp for
  for (int i=0; i<size; i++) {
    a[i] = b[i] + c[i] * d[i];
  }
/* global synchronization */
}
}
```

Every single synchronization in this setup costs in the order of **60000 cycles**!

Setup:
32 threads running on a dual socket 8-core SandyBridge-EP

gcc 4.7.0
Why hardware should not be exposed

Such an approach is not portable …

Hardware issues frequently change …

Those nasty hardware details are too difficult to learn for the average programmer …

Important fundamental concepts are stable and portable (ILP, SIMD, memory organization). The basic principals are simple to understand and every programmer should know them.
Approaches to performance optimization

- Trial and error
- Blind data driven
- Automated expert tools
- Highly skilled experts

Highly complex
Problem centric
Tool centric
Focus on resource utilization

1. Instruction execution
   Primary resource of the processor.

2. Data transfer bandwidth
   Data transfers as a consequence of instruction execution.

What is the **limiting resource**?
Do you fully **utilize** available **resources**?
What needs to be done on one slide

- Reduce computational work
- Reduce data volume (over slow data paths)

- Make use of parallel resources
  - Load balancing
  - Serial fraction

- Identify relevant bottleneck(s)
  - Eliminate bottleneck
  - Increase resource utilization

**Final Goal:** Fully exploit offered resources for your specific code!
HARDWARE OPTIMIZATIONS FOR SINGLE-CORE EXECUTION

- ILP
- SIMD
- SMT
- Memory hierarchy
Common technologies

- Instruction Level Parallelism (ILP)
  - Instruction pipelining
  - Superscalar execution
  - Out-of-order execution

- Memory Hierarchy

- Branch Prediction Unit, Hardware Prefetching

- Single Instruction Multiple Data (SIMD)

- Simultaneous Multithreading (SMT)
Multi-Core: Intel Xeon 2600 (2012)

- Xeon 2600 “Sandy Bridge EP”: 8 cores running at 2.7 GHz (max 3.2 GHz)
- Simultaneous Multithreading → reports as 16-way chip
- **2.3 Billion Transistors / 32 nm**
- **Die size: 435 mm²**

2-socket server
General-purpose cache based microprocessor core

- Implements “Stored Program Computer” concept (Turing 1936)
- Similar designs on all modern systems
- (Still) multiple potential bottlenecks

Stored-program computer
Pipelining of arithmetic/functional units

- **Idea:**
  - Split complex instruction into several simple / fast steps (stages)
  - Each step takes the same amount of time, e.g. a single cycle
  - Execute different steps on different instructions at the same time (in parallel)

- **Allows for shorter cycle times** (simpler logic circuits), e.g.:
  - floating point multiplication takes 5 cycles, but
  - processor can work on 5 different multiplications simultaneously
  - one result at each cycle after the pipeline is full

- **Drawback:**
  - Pipeline must be filled - startup times (#Instructions >> pipeline steps)
  - Efficient use of pipelines requires large number of independent instructions → instruction level parallelism
  - Requires complex instruction scheduling by compiler/hardware – software-pipelining / out-of-order

- Pipelining is *widely used* in modern computer architectures
5-stage Multiplication-Pipeline: 
A(i) = B(i) * C(i) ; i=1,...,N

First result is available after 5 cycles (=latency of pipeline)!

Wind-up/-down phases: Empty pipeline stages
Pipelining: The Instruction pipeline

- Besides arithmetic & functional unit, instruction execution itself is pipelined also, e.g.: one instruction performs at least 3 steps:

  Fetch Instruction from L1I → Decode instruction → Execute Instruction

Hardware Pipelining on processor (all units can run concurrently):

- Branches can stall this pipeline! (Speculative Execution, Predication)
- Each unit is pipelined itself (e.g., Execute = Multiply Pipeline)
Superscalar Processors – Instruction Level Parallelism

- Multiple units enable use of Instruction Level Parallelism (ILP): Instruction stream is “parallelized” on the fly

- Issuing \( m \) concurrent instructions per cycle: \( m \)-way superscalar

- Modern processors are 3- to 6-way superscalar & can perform 2 or 4 floating point operations per cycles
Core details: Simultaneous multi-threading (SMT)
Core details: SIMD processing

Single Instruction Multiple Data (SIMD) allows the concurrent execution of the same operation on “wide” registers.

- **SSE**: register width = 128 Bit → 2 DP floating point operands
- **AVX**: register width = 256 Bit → 4 DP floating point operands

Adding two registers holding double precision floating point operands

**Scalar execution:**

\[ R2 \leftarrow \text{ADD} [R0,R1] \]

**SIMD execution:**

\[ \text{V64ADD} [R0,R1] \rightarrow R2 \]
SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n;i++)
    C[i]=A[i]+B[i];
```

“Loop unrolling”

```
for(int i=0; i<n;i+=4)
{
    C[i] =A[i] +B[i];
    C[i+1]=A[i+1]+B[i+1];
}
```

//remainder loop handling

```
LABEL1:
VLOAD R0  A[i]
VLOAD R1  B[i]
V64ADD[R0,R1] → R2
VSTORE R2 → C[i]
i←i+4
i<(n-4)? JMP LABEL1
```

//remainder loop handling

Load 256 Bits starting from address of A[i] to register R0

Add the corresponding 64 Bit entries in R0 and R1 and store the 4 results to R2

Store R2 (256 Bit) to address starting at C[i]
SIMD processing – Basics

No SIMD vectorization for loops with data dependencies:

```c
for(int i=0; i<n;i++)
    A[i]=A[i-1]*s;
```

“Pointer aliasing” may prevent SIMDification

```c
void f(double *A, double *B, double *C, int n) {
    for(int i=0; i<n; ++i)
        C[i] = A[i] + B[i];
}
```

C/C++ allows that $A \rightarrow &C[-1]$ and $B \rightarrow &C[-2]$
$\rightarrow C[i] = C[i-1] + C[i-2]:$ dependency $\rightarrow$ No SIMD

If “pointer aliasing” is not used, tell it to the compiler:

- `fno-alias` (Intel), `-Msafeptr` (PGI), `-fargument-noalias` (gcc)
- `restrict` keyword (C only!):

```c
void f(double restrict *A, double restrict *B, double restrict *C, int n) {...}
```
Why and how?

Why check the assembly code?

- Sometimes the only way to make sure the compiler “did the right thing”
  - Example: “LOOP WAS VECTORIZED” message is printed, but Loads & Stores may still be scalar!
- Get the assembler code (Intel compiler):
  ```
  icc -S -O3 -xHost triad.c -o a.out
  ```
- Disassemble Executable:
  ```
  objdump -d ./a.out | less
  ```

The x86 ISA is documented in:

- Intel Software Development Manual (SDM) 2A and 2B
Basics of the x86-64 ISA

- Instructions have 0 to 2 operands
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two syntax forms: Intel (left) and AT&T (right)
- Addressing Mode: BASE + INDEX * SCALE + DISPLACEMENT
- C: A[i] equivalent to *(A+i) (a pointer has a type: A+i*8)

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```

```
movaps %xmm4, 48(%rdi,%rax,8)
addq $8, %rax
js ..B1.4
```

```
401b9f: 0f 29 5c c7 30 movaps %xmm3,0x30(%rdi,%rax,8)
401ba4: 48 83 c0 08 add $0x8,%rax
401ba8: 78 a6 js 401b50 <triad_asm+0x4b>
```
Basics of the x86-64 ISA

16 general Purpose Registers (*64bit*):
rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15
alias with eight 32 bit register set:
eax, ebx, ecx, edx, esi, edi, esp, ebp

Floating Point **SIMD** Registers:
xmm0–xmm15  SSE (128bit)  alias with 256-bit registers
ymm0–ymm15  AVX (256bit)

**SIMD** instructions are distinguished by:
**AVX** (VEX) prefix:  \( v \)
Operation:  \( \text{mul, add, mov} \)
Modifier:  \( \text{nontemporal (nt), unaligned (u), aligned (a), high (h)} \)
Width:  \( \text{scalar (s), packed (p)} \)
Data type:  \( \text{single (s), double (d)} \)
Case Study: Simplest code for the summation of the elements of a vector (single precision)

```c
float sum = 0.0;

for (int i=0; i<size; i++){
    sum += data[i];
}
```

**To get object code use**
```bash
objdump -d on object file or executable or compile with -S
```

**AT&T syntax:**
```assembly
addss 0(%rdx,%rax,4),%xmm0
```

Instruction code:
- `401d08`: `f3 0f 58 04 82`  
  `addss xmm0,[rdx + rax * 4]`
- `401d0d`: `48 83 c0 01`  
  `add rax,1`
- `401d11`: `39 c7`  
  `cmp edi,eax`
- `401d13`: `77 f3`  
  `ja 401d08`

(final sum across xmm0 omitted)
Latency and bandwidth in modern computer environments

Avoiding slow data paths is the key to most performance optimizations!
How does data travel from memory to the CPU and back?

Remember: Caches are organized in cache lines (e.g., 64 bytes)
Only complete cache lines are transferred between memory hierarchy levels (except registers)

**MISS**: Load or store instruction does not find data in a cache level → CL transfer required

Example: Array copy $A( :) = C( :)$
Recap: Data transfers in a memory hierarchy

- How does data travel from memory to the CPU and back?
- Example: Array copy \( A(:) = C(:) \)
Consequences for data structure layout

- Promote temporal and spatial locality
- Enable packed (block wise) load/store of data
- Memory locality (placement)
- Avoid false cache line sharing
- Access data in long streams to enable efficient latency hiding

Above requirements may collide with object oriented programming paradigm: array of structures vs structure of arrays
Conclusions about core architectures

- All efforts are targeted on increasing **instruction throughput**
- Every hardware optimization puts an **assumption** against the executed software
- One can distinguish transparent and **explicit** solutions

Common technologies:
- Instruction level parallelism (**ILP**)
- Data parallel execution (**SIMD**), does not affect instruction throughput
- Exploit temporal data access locality (**Caches**)
- Hide data access latencies (**Prefetching**)
- Avoid hazards
PRELUDE:
SCALABILITY 4 THE WIN!
Scalability Myth: Code scalability is the key issue

Lore 1
In a world of highly parallel computer architectures only highly scalable codes will survive

Lore 2
Single core performance no longer matters since we have so many of them and use scalable codes
Scalability Myth: Code scalability is the key issue

```c
!$OMP PARALLEL DO
do k = 1 , Nk
    do j = 1 , Nj; do i = 1 , Ni
        y(i,j,k) = b* ( x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) + 
                      x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1) )
    enddo; enddo
enddo
!$OMP END PARALLEL DO
```

Changing only the compile options makes this code scalable on an 8-core chip

![Graph showing speed-up vs. number of cores]

3D Stencil Update ("Jacobi")

- Prepared for the highly parallel era!
  - `-O0`
  - `-O3  -xAVX`
Scalability Myth: Code scalability is the key issue

```c
!$OMP PARALLEL DO
    do k = 1 , Nk
        do j = 1 , Nj; do i = 1 , Ni
            y(i,j,k) = b* ( x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) + 
                              x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1) )
        enddo; enddo
    enddo
!$OMP END PARALLEL DO
```

Upper limit from simple performance model:
35 GB/s & 24 Byte/update
TOPOLOGY OF MULTI-CORE / MULTI-SOCKET SYSTEMS

- Chip Topology
- Node Topology
- Memory Organisation
Timeline of technology developments

- Deep pipeline \(\rightarrow\) High clock
- SSE2
- Deep pipeline \(\rightarrow\) High clock
- SSE2
- Dual Core
- Pentium D 3.6
- Intel Core 2 Duo 3.0
- Core 2 Quad 3.4
- Sandy Bridge 2.9
- Ivy Bridge 2.7
- Westmere 2.93
- Nehalem 3.2
- Octa-core AVX
- 12C
- 3-channel, DDR3 on-chip ccNUMA
Building blocks for multi-core compute nodes

- **Core**: Unit reading and executing instruction stream
- **Chip**: One integrated circuit die
- **Socket/Package**: May consist of multiple chips

- Memory Hierarchy:
  - Private caches
  - Shared caches
  - **ccNUMA**: Replicated memory interfaces
Chip Topologies

- Separation into core and uncore
- Memory hierarchy holding together the chip design
- L1 (L2) private caches
- L3 cache shared (LLC)

- Serialized LLC $\rightarrow$ not scalable

- Segmented ring bus, distributed LLC $\rightarrow$ scalable design
From UMA to ccNUMA Memory architectures

Yesterday (2006): Dual-socket Intel “Core2” node:

- Uniform Memory Architecture (UMA)
- Flat memory ; symmetric MPs

Today: Dual-socket Intel (Westmere,...) node:

- Cache-coherent Non-Uniform Memory Architecture (ccNUMA)
- HT / QPI provide scalable bandwidth at the price of ccNUMA architectures: Where does my data finally end up?
ccNUMA
4 chips, two sockets, 8 threads per ccNUMA domain

ccNUMA map: Bandwidth penalties for remote access

- Run 8 threads per ccNUMA domain (1 chip)
- Place memory in different domain → 4x4 combinations

![Diagram of ccNUMA map with bandwidth penalties](image)
"Golden Rule" of ccNUMA:
A memory page gets mapped into the local memory of the processor that first touches it!

- Except if there is not enough local memory available

**Caveat:** "touch" means "write", not "allocate"

Example:

```c
double *huge = (double*) malloc(N*sizeof(double));
for(i=0; i<N; i++) // or i+=PAGE_SIZE
    huge[i] = 0.0;
```

It is sufficient to touch a single item to map the entire page
The curse and blessing of interleaved placement: OpenMP STREAM on a Cray XE6 Interlagos node

Parallel init: Correct parallel initialization
LD0: Force data into LD0 via `numactl -m 0`
Interleaved: `numactl --interleave <LD range>`
The curse and blessing of interleaved placement: same on 4-socket (48 core) Magny Cours node

![Bandwidth vs. # NUMA domains (6 threads per domain)]
The driving forces behind performance

\[ P = n_{\text{core}} \times F \times S \times \nu \]

<table>
<thead>
<tr>
<th></th>
<th>Intel IvyBridge-EP</th>
<th>IBM Power7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of cores</strong></td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td><strong>FP instructions per cycle</strong> F</td>
<td>2</td>
<td>2 (DP) / 1 (SP)</td>
</tr>
<tr>
<td><strong>FP ops per instructions</strong> S</td>
<td>4 (DP) / 8 (SP)</td>
<td>2 (DP) / 4 (SP) - FMA</td>
</tr>
<tr>
<td><strong>Clock speed [GHz]</strong> ( \nu )</td>
<td>2.7</td>
<td>3.7</td>
</tr>
<tr>
<td><strong>Performance [GF/s]</strong> ( P )</td>
<td>259 (DP) / 518 (SP)</td>
<td>236 (DP/SP)</td>
</tr>
</tbody>
</table>

TOP500 rank 1 (1996)

But: \( P = 5.4 \text{ GF/s} \) or 14.8 GF/s(dp) for serial, non-SIMD code
Parallel programming models on modern compute nodes

- **Shared-memory (intra-node)**
  - Good old MPI (current standard: 3.0)
  - OpenMP (current standard: 4.0)
  - POSIX threads
  - Intel Threading Building Blocks (TBB)
  - Cilk+, OpenCL, StarSs, … you name it
- **“Accelerated”**
  - OpenMP 4.0
  - CUDA
  - OpenCL
  - OpenACC
- **Distributed-memory (inter-node)**
  - MPI (current standard: 3.0)
  - PVM (gone)
- **Hybrid**
  - Pure MPI + X, X == <you name it>

All models require awareness of **topology** and **affinity** issues for getting best performance out of the machine!
Parallel programming models: 

*Pure MPI*

- Machine structure is invisible to user:
  - → Very simple programming model
  - → MPI “knows what to do”!? 
- Performance issues
  - Intranode vs. internode MPI
  - Node/system topology
Parallel programming models:

*Pure threading on the node*

- Machine structure is invisible to user
  - → Very simple programming model
  - Threading SW (OpenMP, pthreads, TBB, …) should know about the details
- Performance issues
  - Synchronization overhead
  - Memory access
  - Node topology

Diagram:
- Master thread
- Fork
- Join
- Parallel region
- Serial region
- Team of threads
- Memory interface
- Coherent link
- Memory
Parallel programming models: Lots of choices

Hybrid MPI+OpenMP on a multicore multisocket cluster

One MPI process / node

One MPI process / socket:
OpenMP threads on same socket: “blockwise”

OpenMP threads pinned “round robin” across cores in node

Two MPI processes / socket
OpenMP threads on same socket
Conclusions about Node Topologies

Modern computer architecture has a rich “topology”

Node-level **hardware parallelism** takes many forms
- Sockets/devices – CPU: 1-8, GPGPU: 1-6
- Cores – moderate (CPU: 4-16) to massive (GPGPU: 1000’s)
- SIMD – moderate (CPU: 2-8) to massive (GPGPU: 10’s-100’s)

Exploiting performance: **parallelism + bottleneck awareness**
- “High Performance Computing” == computing at a bottleneck

**Performance of programs** is sensitive to architecture
- Topology/affinity influences overheads of popular programming models
- Standards do not contain (many) topology-aware features
  - Things are starting to improve slowly (MPI 3.0, OpenMP 4.0)
- Apart from overheads, performance features are largely independent of the programming model
INTERLUDE:
A GLANCE AT CURRENT ACCELERATOR TECHNOLOGY
NVIDIA Kepler GK110 Block Diagram

Architecture

- 7.1B Transistors
- 15 “SMX” units
  - 192 (SP) “cores” each
- > 1 TFLOP DP peak
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3

- 3:1 SP:DP performance

© NVIDIA Corp. Used with permission.
Intel Xeon Phi block diagram

Architecture

- 3B Transistors
- 60+ cores
- 512 bit SIMD
- ≈ 1 TFLOP DP peak
- 0.5 MB L2/core
- GDDR5
- 2:1 SP:DP performance

64 byte/cy
Comparing accelerators

**Intel Xeon Phi**

- **60+ IA32 cores** each with 512 Bit SIMD FMA unit → **480/960 SIMD DP/SP tracks**
  - Clock Speed: ~1000 MHz
  - Transistor count: ~3 B (22nm)
  - Power consumption: ~250 W

- Peak Performance (DP): ~ 1 TF/s
- Memory BW: ~250 GB/s (GDDR5)

- Threads to execute: 60-240+
- Programming:
  - Fortran/C/C++ +OpenMP + SIMD

**NVIDIA Kepler K20**

- **15 SMX units** each with 192 “cores” → **960/2880 DP/SP “cores”**
  - Clock Speed: ~700 MHz
  - Transistor count: 7.1 B (28nm)
  - Power consumption: ~250 W

- Peak Performance (DP): ~ 1.3 TF/s
- Memory BW: ~ 250 GB/s (GDDR5)

- Threads to execute: 10,000+
- Programming:
  - CUDA, OpenCL, (OpenACC)

**TOP7: “Stampede” at Texas Center for Advanced Computing**

**TOP500 rankings Nov 2012**

**TOP1: “Titan” at Oak Ridge National Laboratory**
Trading single thread performance for parallelism: 
*GPGPUs vs. CPUs*

**GPU vs. CPU**

light speed estimate:

1. Compute bound: 2-10x
2. Memory Bandwidth: 1-5x

<table>
<thead>
<tr>
<th></th>
<th>Intel Core i5 – 2500 (“Sandy Bridge”)</th>
<th>Intel Xeon E5-2680 DP node (“Sandy Bridge”)</th>
<th>NVIDIA K20x (“Kepler”)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cores@Clock</strong></td>
<td>4 @ 3.3 GHz</td>
<td>2 x 8 @ 2.7 GHz</td>
<td>2880 @ 0.7 GHz</td>
</tr>
<tr>
<td><em><em>Performance</em>/core</em>*</td>
<td>52.8 GFlop/s</td>
<td>43.2 GFlop/s</td>
<td>1.4 GFlop/s</td>
</tr>
<tr>
<td><strong>Threads@STREAM</strong></td>
<td>&lt;4</td>
<td>&lt;16</td>
<td>&gt;8000?</td>
</tr>
<tr>
<td><strong>Total performance</strong></td>
<td>210 GFlop/s</td>
<td>691 GFlop/s</td>
<td>4,000 GFlop/s</td>
</tr>
<tr>
<td><strong>Stream BW</strong></td>
<td>18 GB/s</td>
<td>2 x 40 GB/s</td>
<td>168 GB/s (ECC=1)</td>
</tr>
<tr>
<td><strong>Transistors / TDP</strong></td>
<td>1 Billion* / 95 W</td>
<td>2 x (2.27 Billion/130W)</td>
<td>7.1 Billion/250W</td>
</tr>
</tbody>
</table>

* Single Precision

* Includes on-chip GPU and PCI-Express

Complete compute device
MULTICORE PERFORMANCE AND TOOLS
PROBING NODE TOPOLOGY

- Standard tools
- likwid-topology
How do we figure out the node topology?

- **Topology** =
  - Where in the machine does core \#n reside? And do I have to remember this awkward numbering anyway?
  - Which cores share which cache levels?
  - Which hardware threads (“logical cores”) share a physical core?

- **Linux**
  - `cat /proc/cpuinfo` is of limited use
  - Core numbers may change across kernels and BIOSes even on identical hardware

- `numactl --hardware` prints ccNUMA node information

- `hwloc` is another option

```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 8189 MB
node 0 free: 3824 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 8192 MB
node 1 free: 28 MB
node 2 cpus: 18 19 20 21 22 23
node 2 size: 8192 MB
node 2 free: 8036 MB
node 3 cpus: 12 13 14 15 16 17
node 3 size: 8192 MB
node 3 free: 7840 MB
```
How do we figure out the node topology?

**LIKWID** tool suite:

Like I Knew What I’m Doing

Open source tool collection (developed at RRZE):
http://code.google.com/p/likwid

http://arxiv.org/abs/1004.4431
Likwid Tool Suite

- Command line tools for Linux:
  - easy to install
  - works with standard linux kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

- Current tools:
  - `likwid-topology`: Print thread and cache topology
  - `likwid-pin`: Pin threaded application without touching code
  - `likwid-perfctr`: Measure performance counters
  - `likwid-powermeter`: Query turbo mode steps. Measure ETS.
  - `likwid-bench`: Low-level bandwidth benchmark generator tool
Output of `likwid-topology -g`
on one node of Cray XE6 “Hermit”

**CPU type:** AMD Interlagos processor

**Hardware Thread Topology**

<table>
<thead>
<tr>
<th>Sockets</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores per socket</td>
<td>16</td>
</tr>
<tr>
<td>Threads per core</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HWThread</th>
<th>Thread</th>
<th>Core</th>
<th>Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Socket 0: ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 )
Socket 1: ( 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 )

**Cache Topology**

| Level | 1 |
| Size | 16 kB |
Output of likwid-topology continued

Level:  2
Size:   2 MB

Level:  3
Size:   6 MB
Cache groups:  ( 0 1 2 3 4 5 6 7 ) ( 8 9 10 11 12 13 14 15 ) ( 16 17 18 19 20 21 22 23 ) ( 24 25 26 27 28 29 30 31 )

**********************************************************************
NUMA Topology
**********************************************************************
NUMA domains: 4

Domain 0:
Processors:  0 1 2 3 4 5 6 7
Memory: 7837.25 MB free of total 8191.62 MB

Domain 1:
Processors:  8 9 10 11 12 13 14 15
Memory: 7860.02 MB free of total 8192 MB

Domain 2:
Processors:  16 17 18 19 20 21 22 23
Memory: 7847.39 MB free of total 8192 MB

Domain 3:
Processors:  24 25 26 27 28 29 30 31
Memory: 7785.02 MB free of total 8192 MB
Output of likwid-topology continued

**********************************************************************************************

Graphical:
**********************************************************************************************

Socket 0:
**********************************************************************************************

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|------------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB |
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| 6MB | | 6MB | |

Socket 1:
**********************************************************************************************

| | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|------------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB |
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| +-----------------------|
| 6MB | | 6MB | |

ENFORCING THREAD/PROCESS-CORE AFFINITY UNDER THE LINUX OS

- Standard tools and OS affinity facilities under program control
- likwid-pin
Example: STREAM benchmark on 16-core Sandy Bridge: Anarchy vs. thread pinning

- No pinning

- There are several reasons for caring about affinity:
  - Eliminating performance variation
  - Making use of architectural features
  - Avoiding resource contention
  - Benchmark how code reacts to variations
More thread/Process-core affinity (“pinning”) options

- Highly OS-dependent system calls
  - But available on all systems
    - Linux: `sched_setaffinity()`
    - Windows: `SetThreadAffinityMask()`
- Open MPI: hwloc library

- Support for “semi-automatic” pinning in some compilers/environments
  - All modern compilers with OpenMP support
  - Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)
  - OpenMP 4.0
  - Affinity awareness in MPI libraries:
    - OpenMPI
    - Intel MPI
    - …
**Likwid-pin**

*Overview*

- Pins processes and threads to specific cores **without touching code**
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Based on combination of wrapper tool together with overloaded pthread library → binary must be dynamically linked!
- Can also be used as a superior **replacement for taskset**
- Supports logical core numbering within a node and within an existing CPU set
  - Useful for running inside CPU sets defined by someone else, e.g., the MPI start mechanism or a batch system
- Usage examples:
  - `likwid-pin -c 0,2,4-6 ./myApp parameters`
  - `likwid-pin -c S0:0-3 ./myApp parameters`
Running the STREAM benchmark with likwid-pin:

```
$ likwid-pin -c 0,1,4,5 ./stream
[likwid-pin] Main PID -> core 0 - OK

Double precision appears to have 16 digits of accuracy
Assuming 8 bytes per DOUBLE PRECISION word

[... some STREAM output omitted ...]

The *best* time for each test is used
*EXCLUDING* the first and last iterations

(pthread wrapper) PIN_MASK: 0->1 1->4 2->5
(pthread wrapper) SKIP MASK: 0x1
(pthread wrapper 0) Notice: Using libpthread.so.0
threadid 1073809728 -> SKIP
(pthread wrapper 1) Notice: Using libpthread.so.0
threadid 1078008128 -> core 1 - OK
(pthread wrapper 2) Notice: Using libpthread.so.0
threadid 1082206528 -> core 4 - OK
(pthread wrapper 3) Notice: Using libpthread.so.0
threadid 1086404928 -> core 5 - OK
[... rest of STREAM output omitted ...]
```
## Likwid-pin

*Using logical core numbering*

- Core numbering may vary from system to system even with identical hardware
  - Likwid-topology delivers this information, which can then be fed into likwid-pin
  - Alternatively, likwid-pin can abstract this variation and provide a purely logical numbering *(physical cores first)*

Across all cores in the node:

\[
\text{likwid-pin} \ -c \ N:0-7 \ ./a.out
\]

Across the cores in each socket and across sockets in each node:

\[
\text{likwid-pin} \ -c \ S0:0-3@S1:0-3 \ ./a.out
\]
Likwid-pin

Using logical core numbering

- Possible unit prefixes
  - N node
  - S socket
  - M NUMA domain
  - C outer level cache group

Default if –c is not specified!
Advanced options for pinning: Expressions

- Expressions are more powerful in situations where the pin mask would be very long or clumsy

Compact pinning:
likwid-pin -c E:<thread domain>:<number of threads>\n    [:<chunk size>:<stride>] ...

Scattered pinning across all domains of the designated type:
likwid-pin -c <domaintype>:scatter

- Examples:
likwid-pin -c E:N:8 ...   # equivalent to N:0-7
likwid-pin -c E:N:120:2:4 ... # Phi: 120 threads, 2 per core

- Scatter across all NUMA domains:
likwid-pin -c M:scatter
Intel KMP_AFFINITY environment variable

- \texttt{KMP\_AFFINITY} = [\texttt{<modifier>,...}]\texttt{<type>},[,\texttt{<permute>}][,\texttt{<offset>}]\

- **modifier**
  - granuality=\texttt{<specifier>} takes the following specifiers: fine, thread, and core
  - norespect
  - noverbose
  - proclist=\texttt{<proc-list>}
  - respect
  - verbose

- **type (required)**
  - compact
  - disabled
  - explicit (\texttt{GOMP\_CPU\_AFFINITY})
  - none
  - scatter

- Default: noverbose, respect, granularity=core

- \texttt{KMP\_AFFINITY=verbose,none} to list machine topology map

**OS processor IDs**

Respect an OS affinity mask in place
Intel KMP_AFFINITY examples

- **KMP_AFFINITY=granularity=fine,compact**

(c) Intel

```
Package 0
  Core 0
  0 1
  0 1

Package 3
  Core 0
  0 1
  0 1
```

Package means chip/socket

- **KMP_AFFINITY=granularity=fine,scatter**

(c) Intel

```
Package 0
  Core 0
  0 1
  0 1

Package 3
  Core 1
  0 1
  0 1
```

OpenMP* global thread IDs

Thread context
GNU GOMP_AFFINITY

- GOMP_AFFINITY=3,0-2 used with 6 threads

- Always operates with OS processor IDs

(c) Intel
PROBING PERFORMANCE BEHAVIOR

likwid-perfctr
likwid-perfctr

Basic approach to performance analysis

1. Runtime profile / Call graph (gprof)
2. Instrument those parts which consume a significant part of runtime
3. Find performance signatures

Possible signatures:
- Bandwidth saturation
- Instruction throughput limitation (real or language-induced)
- Latency impact (irregular data access, high branch ratio)
- Load imbalance
- ccNUMA issues (data access across ccNUMA domains)
- Pathologic cases (false cacheline sharing, expensive operations)
Probing performance behavior

- How do we find out about the performance properties and requirements of a parallel code?
  - Profiling via advanced tools is often overkill
  - A coarse overview is often sufficient
    - \texttt{likwid-perfctr} (similar to “perfex” on IRIX, “hpmcount” on AIX, “lipfpm” on Linux/Altix)

- Simple end-to-end measurement of hardware performance metrics
- “Marker” API for starting/stopping counters
- Multiple measurement region support
- Preconfigured and extensible metric groups, list with \texttt{likwid-perfctr -a}

\begin{itemize}
  \item BRANCH: Branch prediction miss rate/ratio
  \item CACHE: Data cache miss rate/ratio
  \item CLOCK: Clock of cores
  \item DATA: Load to store ratio
  \item FLOPS\_DP: Double Precision MFlops/s
  \item FLOPS\_SP: Single Precision MFlops/s
  \item FLOPS\_X87: X87 MFlops/s
  \item L2: L2 cache bandwidth in MBytes/s
  \item L2CACHE: L2 cache miss rate/ratio
  \item L3: L3 cache bandwidth in MBytes/s
  \item L3CACHE: L3 cache miss rate/ratio
  \item MEM: Main memory bandwidth in MBytes/s
  \item TLB: TLB miss rate/ratio
\end{itemize}
Example usage with preconfigured metric group

```bash
$ env OMP_NUM_THREADS=4 likwid-perfctr -C N:0-3 -g FLOPS_DP ./stream.exe
```

CPU type: Intel Core Lynnfield processor
CPU clock: 2.93 GHz

Measuring group FLOPS_DP

YOUR PROGRAM OUTPUT

<table>
<thead>
<tr>
<th>Event</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>1.97463e+08</td>
<td>2.31001e+08</td>
<td>2.30963e+08</td>
<td>2.31885e+08</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>9.56999e+08</td>
<td>9.58401e+08</td>
<td>9.58637e+08</td>
<td>9.57338e+08</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_PACKED</td>
<td>4.00294e+07</td>
<td>3.08927e+07</td>
<td>3.08866e+07</td>
<td>3.08904e+07</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_SCALAR</td>
<td>882</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION</td>
<td>4.00303e+07</td>
<td>3.08927e+07</td>
<td>3.08866e+07</td>
<td>3.08904e+07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [s]</td>
<td>0.326242</td>
<td>0.32672</td>
<td>0.326801</td>
<td>0.326358</td>
</tr>
<tr>
<td>CPI</td>
<td>4.84647</td>
<td>4.14891</td>
<td>4.15061</td>
<td>4.12849</td>
</tr>
<tr>
<td>DP MFlops/s (DP assumed)</td>
<td>245.399</td>
<td>189.108</td>
<td>189.024</td>
<td>189.304</td>
</tr>
<tr>
<td>Packed MUOPS/s</td>
<td>122.698</td>
<td>94.554</td>
<td>94.5121</td>
<td>94.6519</td>
</tr>
<tr>
<td>Scalar MUOPS/s</td>
<td>0.00270351</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SP MUOPS/s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DP MUOPS/s</td>
<td>122.701</td>
<td>94.554</td>
<td>94.5121</td>
<td>94.6519</td>
</tr>
</tbody>
</table>

Derived metrics

Always measured

Configured metrics (this group)
likwid-perfctr
Marker API

- A marker API is available to restrict measurements to code regions
- The API only turns counters on/off. The configuration of the counters is still done by likwid-perfctr
- Multiple named regions support, accumulation over multiple calls
- Inclusive and overlapping regions allowed

```c
#include <likwid.h>
...
LIKWARD_MARKER_INIT; // must be called from serial region
#pragma omp parallel
{
 LIKWARD_MARKER_THREADINIT; // only reqd. if measuring multiple threads
}
...
LIKWARD_MARKER_START("Compute");
...
LIKWARD_MARKER_STOP("Compute");
...
LIKWARD_MARKER_START("Postprocess");
...
LIKWARD_MARKER_STOP("Postprocess");
...
LIKWARD_MARKER_CLOSE; // must be called from serial region
```

Activate macros with `-DLIKWARD_PERFMON`
Basics of Benchmarking
Performance Patterns
Signatures
1. Define relevant test cases
2. Establish a sensible performance metric
3. Acquire a runtime profile (sequential)
4. Identify hot kernels (Hopefully there are any!)
5. Carry out optimization process for each kernel

Motivation:
- Understand observed performance
- Learn about code characteristics and machine capabilities
- Deliberately decide on optimizations
Best Practices Benchmarking

Preparation
- Reliable timing (Minimum time which can be measured?)
- Document code generation (Flags, Compiler Version)
- Get exclusive System
- System state (Clock, Turbo mode, Memory, Caches)
- Consider to automate runs with a skript (Shell, python, perl)

Doing
- Affinity control
- Check: Is the result reasonable?
- Is result deterministic and reproducible.
- Statistics: Mean, Best ??
- Basic variants: Thread count, affinity, working set size (Baseline!)
Best Practices Benchmarking cont.

Postprocessing

- Documentation
- Try to understand and explain the result
- Plan variations to gain more information
- Many things can be better understood if you plot them (gnuplot, xmgrace)
Thinking in Bottlenecks

- A bottleneck is a performance limiting setting
- Microarchitectures expose numerous bottlenecks

Observation 1:
Most applications face a single bottleneck at a time!

Observation 2:
There is a limited number of relevant bottlenecks!
Process vs. Tool

Reduce complexity!

We propose a human driven process to enable a systematic way to success!

- Executed by humans.
- Uses tools by means of data acquisition only.

Uses one of the most powerful tools available: Your brain!

You are a investigator making sense of what’s going on.
Performance Engineering Process: Analysis

Step 1 **Analysis**: Understanding observed performance

The set of input data indicating a pattern is its **signature**

**Performance patterns** are typical performance limiting motifs

- Algorithm/Code Analysis
- Hardware/Instruction set architecture
- Microbenchmarking
- Application Benchmarking

Pattern
Performance analysis phase

Understand observed performance: **Where am I?**

**Input:**
- Static code analysis
- HPM data
- Scaling data set size
- Scaling number of used cores
- Microbenchmarking

**Performance patterns** are typical performance limiting motives. The set of input data indicating a pattern is its **signature**.
Performance Engineering Process: Modelling

Step 2 **Formulate Model**: Validate pattern and get quantitative insight.
Step 3 **Optimization**: Improve utilization of offered resources.

- **Optimize for better resource utilization**
- **Eliminate non-expedient activity**

- Pattern
- Performance Model

Performance improves until next bottleneck is hit.
Performance pattern classification

1. Maximum resource utilization
2. Hazards
3. Work related (Application or Processor)

The system offers two basic resources:

- **Execution of instructions** (primary)
- **Transferring data** (secondary)
## Patterns (I): Bottlenecks & hazards

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Performance behavior</th>
<th>Metric signature, LIKWID performance group(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bandwidth saturation</strong></td>
<td>Saturating speedup across cores sharing a data path</td>
<td>Bandwidth meets BW of suitable streaming benchmark (MEM, L3)</td>
</tr>
<tr>
<td><strong>ALU saturation</strong></td>
<td>Throughput at design limit(s)</td>
<td>Good (low) CPI, integral ratio of cycles to specific instruction count(s) (FLOPS*, DATA, CPI)</td>
</tr>
<tr>
<td><strong>Inefficient data access</strong></td>
<td>Simple bandwidth performance model much too optimistic</td>
<td>Low BW utilization / Low cache hit ratio, frequent CL evicts or replacements (CACHE, DATA, MEM)</td>
</tr>
<tr>
<td>Latency-bound access</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Micro-architectural anomalies</strong></td>
<td>Large discrepancy from simple performance model based on LD/ST and arithmetic throughput</td>
<td>Relevant events are very hardware-specific, e.g., memory aliasing stalls, conflict misses, unaligned LD/ST, requeue events</td>
</tr>
</tbody>
</table>
### Patterns (II): Hazards

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Performance behavior</th>
<th>Metric signature, LIKwid performance group(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>False sharing of cache lines</strong></td>
<td>Large discrepancy from performance model in parallel case, bad scalability</td>
<td>Frequent (remote) CL evicts (CACHE)</td>
</tr>
<tr>
<td><strong>Bad ccNUMA page placement</strong></td>
<td>Bad or no scaling across NUMA domains, performance improves with interleaved page placement</td>
<td>Unbalanced bandwidth on memory interfaces / High remote traffic (MEM)</td>
</tr>
<tr>
<td><strong>Pipelining issues</strong></td>
<td>In-core throughput far from design limit, performance insensitive to data set size</td>
<td>(Large) integral ratio of cycles to specific instruction count(s), bad (high) CPI (FLOPS_*, DATA, CPI)</td>
</tr>
<tr>
<td><strong>Control flow issues</strong></td>
<td>See above</td>
<td>High branch rate and branch miss ratio (BRANCH)</td>
</tr>
</tbody>
</table>
## Patterns (III): Work-related

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Performance behavior</th>
<th>Metric signature, LIKWID performance group(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load imbalance / serial fraction</td>
<td>Saturating/sub-linear speedup</td>
<td>Different amount of “work” on the cores (FLOPS_*); note that instruction count is not reliable!</td>
</tr>
<tr>
<td>Synchronization overhead</td>
<td>Speedup going down as more cores are added / No speedup with small problem sizes / Cores busy but low FP performance</td>
<td>Large non-FP instruction count (growing with number of cores used) / Low CPI (FLOPS_*, CPI)</td>
</tr>
<tr>
<td>Instruction overhead</td>
<td>Low application performance, good scaling across cores, performance insensitive to problem size</td>
<td>Low CPI near theoretical limit / Large non-FP instruction count (constant vs. number of cores) (FLOPS_*, DATA, CPI)</td>
</tr>
<tr>
<td>Code composition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expensive instructions</td>
<td>Similar to instruction overhead</td>
<td>Many cycles per instruction (CPI) if the problem is large-latency arithmetic</td>
</tr>
<tr>
<td>Ineffective instructions</td>
<td></td>
<td>Scalar instructions dominating in data-parallel loops (FLOPS_*, CPI)</td>
</tr>
</tbody>
</table>
Example rabbitCT

Result of effort:
5-6 x improvement against initial parallel C code implementation
>50% of peak performance (SSE)

ECM Model analysis using IACA

ALU saturation, Pipelining issues, Code composition patterns
Replace divide with pipelined reciprocal
Apply SIMD vectorization
Use SMT capabilities

ALU saturation pattern
Optimization without knowledge about bottleneck

![Graph showing optimization without knowledge about bottleneck]
Where to start

Look at the code and understand what it is doing!

Scaling runs:
- Scale #cores inside ccNUMA domain
- Scale across ccNUMA domains
- Scale working set size (if possible)

HPM measurements:
- Memory Bandwidth
- Instruction decomposition: Arithmetic, data, branch, other
- SIMD vectorized fraction
- Data volumes inside memory hierarchy
- CPI
Most frequent patterns (seen with scientific computing glasses)

Data transfer related:
- Memory bandwidth saturation
- Bad ccNUMA page placement

Parallelization
- Load imbalance
- Serial fraction

Code composition:
- Instruction overhead
- Ineffective instructions
- Expensive instructions

Overhead:
- Synchronization overhead

Excess work:
- Data volume reduction over slow data paths
- Reduction of algorithmic work
**Pattern: Bandwidth Saturation**

1. Perform scaling run inside ccNUMA domain
2. Measure memory bandwidth with HPM
3. Compare to micro benchmark with similar data access pattern

![Graph showing scalable and saturating bandwidth](image)

- Measured bandwidth `spmv`: 45964 MB/s
- Synthetic load benchmark: 47022 MB/s
Consequences from the saturation pattern

Clearly distinguish between “saturating” and “scalable” performance on the chip level

![Graph showing saturating and scalable performance](image)
Consequences from the saturation pattern

There is no clear bottleneck for single-core execution.

Code profile for single thread ≠ code profile for multiple threads

→ Single-threaded profiling may be misleading

1 thread

| saturating part | scalable part |

8 threads

runtime
Pattern: Load imbalance

1. Check HPM instruction count distribution across cores
   - Instructions retired / CPI may not be a good indication of useful workload – at least for numerical / FP intensive codes....
   - Floating Point Operations Executed is often a better indicator

```
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, I
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```
Example for a load balanced code

```c
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, N
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```

Higher CPI but better performance

```
env OMP_NUM_THREADS=6 likwid-perfctr -C S0:0-5 -g FLOPS_DP ./a.out
```
Pattern: Bad ccNUMA page placement

1. Benchmark scaling across ccNUMA domains
2. Is performance sensitive to interleaved page placement
3. Measure inter-socket traffic with HPM
Pattern: Instruction Overhead

1. Perform a HPM instruction decomposition analysis
2. Measure resource utilization
3. Static code analysis

<table>
<thead>
<tr>
<th>Instruction decomposition</th>
<th>Inlining failed</th>
<th>Inefficient data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic FP</td>
<td>12%</td>
<td>21%</td>
</tr>
<tr>
<td>Load/Store</td>
<td>30%</td>
<td>50%</td>
</tr>
<tr>
<td>Branch</td>
<td>24%</td>
<td>10%</td>
</tr>
<tr>
<td>Other</td>
<td>34%</td>
<td>19%</td>
</tr>
</tbody>
</table>

C++ codes which suffer from overhead (inlining problems, complex abstractions) need a lot more overall instructions related to the arithmetic instructions

- Often (but not always) “good” (i.e., low) CPI
- Low-ish bandwidth
- Low # of floating-point instructions vs. other instructions
Pattern: Inefficient Instructions

1. HPM measurement: Relation packed vs. scalar instructions
2. Static assembly code analysis: Search for scalar loads

There is usually no counter for packed vs scalar (SIMD) loads and stores.
Also the compiler usually does not distinguish!

Only solution: Inspect code at assembly level.
Pattern: Synchronization overhead

1. Performance is decreasing with growing core counts
2. Performance is sensitive to topology
3. Static code analysis: Estimate work vs. barrier cost.
## Thread synchronization overhead on SandyBridge-EP

*Barrier overhead in CPU cycles*

<table>
<thead>
<tr>
<th>2 Threads</th>
<th>Intel 13.1.0</th>
<th>GCC 4.7.0</th>
<th>GCC 4.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared L3</td>
<td>384</td>
<td>5242</td>
<td>4616</td>
</tr>
<tr>
<td>SMT threads</td>
<td>2509</td>
<td>3726</td>
<td>3399</td>
</tr>
<tr>
<td>Other socket</td>
<td>1375</td>
<td>5959</td>
<td>4909</td>
</tr>
</tbody>
</table>

 GCC not very competitive

<table>
<thead>
<tr>
<th>Full domain</th>
<th>Intel 13.1.0</th>
<th>GCC 4.7.0</th>
<th>GCC 4.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>1497</td>
<td>14546</td>
<td>14418</td>
</tr>
<tr>
<td>Node</td>
<td>3401</td>
<td>34667</td>
<td>29788</td>
</tr>
<tr>
<td>Node +SMT</td>
<td>6881</td>
<td>59038</td>
<td>58898</td>
</tr>
</tbody>
</table>
## Thread synchronization overhead on AMD Interlagos

### Barrier overhead in CPU cycles

<table>
<thead>
<tr>
<th></th>
<th>Cray 8.03</th>
<th>GCC 4.6.2</th>
<th>PGI 11.8</th>
<th>Intel 12.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2 Threads</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared L2</td>
<td>258</td>
<td>3995</td>
<td>1503</td>
<td>128623</td>
</tr>
<tr>
<td>Shared L3</td>
<td>698</td>
<td>2853</td>
<td>1076</td>
<td>128611</td>
</tr>
<tr>
<td>Same socket</td>
<td>879</td>
<td>2785</td>
<td>1297</td>
<td>128695</td>
</tr>
<tr>
<td>Other socket</td>
<td><strong>940</strong></td>
<td><strong>2740 / 4222</strong></td>
<td>1284 / 1325</td>
<td><strong>128718</strong></td>
</tr>
</tbody>
</table>

### Intel compiler barrier very expensive on Interlagos

OpenMP & Cray compiler 😊

### Full domain

<table>
<thead>
<tr>
<th></th>
<th>Cray 8.03</th>
<th>GCC 4.6.2</th>
<th>PGI 11.8</th>
<th>Intel 12.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shared L3</strong></td>
<td>2272</td>
<td>27916</td>
<td>5981</td>
<td>151939</td>
</tr>
<tr>
<td><strong>Socket</strong></td>
<td>3783</td>
<td>49947</td>
<td>7479</td>
<td>163561</td>
</tr>
<tr>
<td><strong>Node</strong></td>
<td>7663</td>
<td>167646</td>
<td>9526</td>
<td>178892</td>
</tr>
</tbody>
</table>
Thread synchronization overhead on Intel Xeon Phi

*Barrier overhead in CPU cycles*

<table>
<thead>
<tr>
<th></th>
<th>SMT1</th>
<th>SMT2</th>
<th>SMT3</th>
<th>SMT4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One core</strong></td>
<td>n/a</td>
<td>1597</td>
<td>2825</td>
<td>3557</td>
</tr>
<tr>
<td><strong>Full chip</strong></td>
<td>10604</td>
<td>12800</td>
<td>15573</td>
<td>18490</td>
</tr>
</tbody>
</table>

That does not look bad for 240 threads!

Still the pain may be much larger, as more work can be done in one cycle on Phi compared to a full Sandy Bridge node

- 3.75 x cores (16 vs 60) on Phi
- 2 x more operations per cycle on Phi
- 2.7 x more barrier penalty (cycles) on Phi

7.5 x more work done on Xeon Phi per cycle

One barrier causes 2.7 x 7.5 = 20x more pain 😊.
SpMV kernel: Data set size and thread count influence on limiting pattern

Strongly memory-bound for large data sets

- Streaming, with partially indirect access:

```fortran
!$OMP parallel do
do i = 1,Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    c(i) = c(i) + val(j) * b(col_idx(j))
  enddo
endo
do
endo
!$OMP end parallel do
```

- Usually many spMVMs required to solve a problem

- Following slides: Performance data on one 24-core AMD Magny Cours node
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

Large matrix

**Pattern:** Bandwidth saturation

Intrasocket bandwidth bottleneck

Good scaling across NUMA domains
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

Medium size

Pattern: Work reduction. Less data volume over slow data paths

Intrasocket bandwidth bottleneck

Working set fits in aggregate cache
Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

Small size

Pattern: Synchronization overhead

No bandwidth bottleneck

Parallelization overhead dominates

rbs480a, 480x480, non-zero: 17088
“SIMPLE” PERFORMANCE MODELING: THE ROOFLINE MODEL

Loop-based performance modeling: Execution vs. data transfer
How to perform a instruction throughput analysis on the example of Intel’s port based scheduler model.

Preliminary: Estimating Instruction throughput

Issue 6 uops

Port 0: ALU
Port 1: ALU
Port 2: LOAD
Port 3: LOAD
Port 4: STORE
Port 5: ALU

FMUL
FADD
AGU
AGU
FSHUF

16b upwards
16b upwards
16b downwards

Retire 4 uops

JUMP

SandyBridge
Every new generation provides incremental improvements. The OOO microarchitecture is a blend between P6 (Pentium Pro) and P4 (Netburst) architectures.

### Preliminary: Estimating Instruction throughput

<table>
<thead>
<tr>
<th>Port 0</th>
<th>Port 1</th>
<th>Port 2</th>
<th>Port 3</th>
<th>Port 4</th>
<th>Port 5</th>
<th>Port 6</th>
<th>Port 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>ALU</td>
<td>LOAD</td>
<td>LOAD</td>
<td>STORE</td>
<td>ALU</td>
<td>ALU</td>
<td>AGU</td>
</tr>
<tr>
<td>FMA</td>
<td>FMA</td>
<td>AGU</td>
<td>AGU</td>
<td></td>
<td>FSHUF</td>
<td>JUMP</td>
<td></td>
</tr>
<tr>
<td>FMUL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>JUMP</td>
<td></td>
</tr>
</tbody>
</table>

- Issue 8 uops
- Retire 4 uops

Haswell
Exercise: Estimate performance of triad on SandyBridge @3GHz

for (int i=0; i<N; i++) {
}

How many cycles to process one 64byte cacheline?

64byte equivalent to 8 scalar iterations or 2 AVX vector iterations.

Cycle 1: load and ½ store and mult and add
Cycle 2: load and ½ store
Cycle 3: load

Answer: 6 cycles
Exercise: Estimate performance of triad on SandyBridge @3GHz

for (int i=0; i<N; i++) {
}

What's the performance in GFlops/s and bandwidth in MBytes/s?

One AVX iteration (3 cycles) performs 4x2=8 flops.

(3 GHZ / 3 cycles) * 4 updates * 2 flops/update = 8 GFlops/s
4 GUPS/s * 4 words/update * 8byte/word = 128 GBytes/s
The Roofline Model\textsuperscript{1,2}

1. \( P_{\text{max}} \) = Applicable peak performance of a loop, assuming that data comes from L1 cache (this is not necessarily \( P_{\text{peak}} \))

2. \( I \) = Computational intensity ("work" per byte transferred) over the slowest data path utilized ("the bottleneck")
   - Code balance \( B_C = I^{-1} \)

3. \( b_S \) = Applicable peak bandwidth of the slowest data path utilized

Expected performance: \( P = \min(P_{\text{max}}, I b_S) \)

\textsuperscript{1} W. Schönauer: *Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers*. (2000)

“Simple” Roofline: The vector triad

Example: **Vector triad** \( A(:\cdot) = B(:\cdot) + C(:\cdot) \cdot D(:\cdot) \)
on a 2.7 GHz 8-core Sandy Bridge chip (AVX vectorized)

- \( b_S = 40 \) GB/s
- \( B_c = (4+1) \) Words / 2 Flops = 2.5 W/F (including write allocate)
  \[ I = 0.4 \text{ F/W} = 0.05 \text{ F/B} \]
  \[ I \cdot b_S = 2.0 \text{ GF/s} \] (1.2 % of peak performance)

- \( P_{\text{peak}} = 173 \text{ GFlop/s} \) (8 FP units x (4+4) Flops/cy x 2.7 GHz)
- \( P_{\text{max}}? \) \( \rightarrow \) Observe LD/ST throughput maximum of 1 AVX Load and \( \frac{1}{2} \) AVX store per cycle \( \rightarrow \) 3 cy / 8 Flops
  \[ P_{\text{max}} = 57.6 \text{ GFlop/s} \] (33% peak)

\[ P = \min(P_{\text{max}}, I \cdot b_S) = \min(57.6, 2.0) \text{GFlop/s} = 2.0 \text{ GFlop/s} \]
A not so simple Roofline example

Example:

```latex
do i=1,N; s=s+a(i); enddo
```

in double precision on a 2.7 GHz Sandy Bridge socket @ “large” N

How do we get these?

- ADD peak (best possible code)
- no SIMD
- 3-cycle latency per ADD if not unrolled

How do we get these?

- See next!

$I = 1$ Flop / 8 byte (in DP)

$P = 5$ Gflop/s
Applicable peak for the summation loop

Plain scalar code, no SIMD

LOAD r1.0 ← 0
i ← 1
loop:
    LOAD r2.0 ← a(i)
    ADD r1.0 ← r1.0+r2.0
    ++i ⇒ loop
result ← r1.0

ADD pipes utilization:

⇒ 1/12 of ADD peak
Applicable peak for the summation loop

Scalar code, 3-way unrolling

LOAD r1.0 ← 0
LOAD r2.0 ← 0
LOAD r3.0 ← 0
i ← 1
loop:
    LOAD r4.0 ← a(i)
    LOAD r5.0 ← a(i+1)
    LOAD r6.0 ← a(i+2)
    ADD r1.0 ← r1.0+r4.0
    ADD r2.0 ← r2.0+r5.0
    ADD r3.0 ← r3.0+r6.0
    i+=3 →? loop
result ← r1.0+r2.0+r3.0

ADD pipes utilization:

→ 1/4 of ADD peak
Applicable peak for the summation loop

SIMD-vectorized, 3-way unrolled

LOAD [r1.0,…,r1.3] ← [0,0]
LOAD [r2.0,…,r2.3] ← [0,0]
LOAD [r3.0,…,r3.3] ← [0,0]
i ← 1

loop:
  LOAD [r4.0,…,r4.3] ← [a(i),…,a(i+3)]
  LOAD [r5.0,…,r5.3] ← [a(i+4),…,a(i+7)]
  LOAD [r6.0,…,r6.3] ← [a(i+8),…,a(i+11)]
  ADD r1 ← r1+r4
  ADD r2 ← r2+r5
  ADD r3 ← r3+r6
  i+=12 ➔? loop

result ← r1.0+r1.1+…+r3.2+r3.3

ADD pipes utilization:
Input to the roofline model

... on the example of

\[
\text{do } i=1,N; \quad s=s+a(i); \quad \text{enddo}
\]

**Throughput:** 1 ADD + 1 LD/cy

**Pipeline depth:** 3 cy (ADD)

4-way SIMD, 8 cores

**Code analysis:**
1 ADD + 1 LOAD

**Memory-bound @ large N!**

\[ P_{\text{max}} = 5 \text{ GF/s} \]

**Maximum memory bandwidth:** 40 GB/s

**Analysis**

**Architecture**

7.2 ... 86.4 GF/s

5 GF/s

measurement
Assumptions for the Roofline Model

The roofline formalism is based on some (crucial) assumptions:

- There is a clear concept of “work” vs. “traffic”
  - “work” = flops, updates, iterations…
  - “traffic” = required data to do “work”

- **Attainable bandwidth of code = input parameter!** Determine effective bandwidth via simple streaming benchmarks to model more complex kernels and applications

- **Data transfer and core execution overlap** perfectly!

- **Slowest data path is modeled only**; all others are assumed to be infinitely fast

- If data transfer is the limiting factor, the **bandwidth** of the slowest data path can be **utilized to 100% (“saturation”)**

- Latency effects are ignored, i.e. **perfect streaming mode**
Typical code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code

2. Increase intensity to make better use of BW bottleneck

3. Increase intensity and go from memory-bound to core-bound

4. Hit the core bottleneck by good serial code

5. Shift $P_{\text{max}}$ by accessing additional hardware features or using a different algorithm/implementation
Shortcomings of the roofline model

Saturation effects in multicore chips are not explained

- Reason: “saturation assumption”
- Cache line transfers and core execution do sometimes not overlap perfectly
- Only increased “pressure” on the memory interface can saturate the bus → need more cores!

ECM model gives more insight
Where the roofline model fails

In memory performance below saturation point
ECM Model

ECM = “Execution-Cache-Memory”

Assumptions:
Single-core execution time is composed of
1. In-core execution
2. Data transfers in the memory hierarchy
Data transfers may or may not overlap with each other or with in-core execution
Scaling is linear until the relevant bottleneck is reached

Input:
Same as for Roofline
+ data transfer times in hierarchy
Introduction to ECM model

ECM = “Execution-Cache-Memory”

• Analytical performance model

• Focus on resource utilization
  • Instruction Execution
  • Data Movement

• Lightspeed assumption:
  • Optimal instruction throughput
  • Always bandwidth bound

The RULES™
1. Single-core execution time is composed of
   1. In-core execution
   2. Data transfers in the memory hierarchy
2. All timings are in units of one CL
3. LOADS in the L1 cache do not overlap with any other data transfer
4. Scaling across cores is linear until a shared bottleneck is hit
Vector dot product: Code characteristics

```c
double sum = 0.0;
for (int i=0; i<N; i++) {
    sum += a[i]*b[i];
}
```

**Naive**

```c
double sum = 0.0;
double c = 0.0;
for (int i=0; i<N; i++) {
    double prod = a[i]*b[i];
    double y = prod-c;
    double t = sum+y;
    c = (t-sum)-y;
    sum = t;
}
```

**Kahan**

<table>
<thead>
<tr>
<th></th>
<th>naive</th>
<th>kahan</th>
</tr>
</thead>
<tbody>
<tr>
<td>loads</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>mul</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
## Machine Model

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>Xeon E5-2680</td>
<td>Xeon E5-2690 v2</td>
<td>Xeon E5-2695 v3</td>
</tr>
<tr>
<td><strong># cores</strong></td>
<td>8 cores @ 2.7GHz</td>
<td>10 cores @ 3.0GHz</td>
<td>14 cores @ 2.3GHz</td>
</tr>
<tr>
<td><strong>Load / Store</strong></td>
<td>2 L + 1 S per cy</td>
<td>2 L + 1 S per cy</td>
<td>2 L + 1 S per cy</td>
</tr>
<tr>
<td><strong>L1 Port Width</strong></td>
<td>16b</td>
<td>16b</td>
<td>32b</td>
</tr>
<tr>
<td><strong>Add</strong></td>
<td>1 per cy</td>
<td>1 per cy</td>
<td>1 per cy</td>
</tr>
<tr>
<td><strong>Mul</strong></td>
<td>1 per cy</td>
<td>1 per cy</td>
<td>2 per cy</td>
</tr>
<tr>
<td><strong>FMA</strong></td>
<td>n/a</td>
<td>n/a</td>
<td>2 per cy</td>
</tr>
<tr>
<td><strong>SIMD width</strong></td>
<td>32b</td>
<td>32b</td>
<td>32b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 – L2</strong></td>
<td>32b/cy 2cy/CL</td>
<td>32b/cy 2cy/CL</td>
<td>64b/cy 1cy/CL</td>
</tr>
<tr>
<td><strong>L2 – L3</strong></td>
<td>32b/cy 2cy/CL</td>
<td>32b/cy 2cy/CL</td>
<td>32b/cy 2cy/CL</td>
</tr>
<tr>
<td><strong>L3 - MEM</strong></td>
<td>4.0cy/CL</td>
<td>3.5cy/CL</td>
<td>2.5cy/CL</td>
</tr>
</tbody>
</table>
Example Kahan (AVX) on IvyBridge-EP

Shorthand notation:
\[
T_{\text{core}} = \max(T_{nOL}, T_{OL}) \]
\[
T_{\text{ECM}} = \max(T_{nOL} + T_{\text{data}}, T_{OL})
\]

Contributions:
\[
\{ T_{OL} \parallel T_{nOL} \parallel T_{L1/L2} \parallel T_{L2/L3} \parallel T_{L3/MEM} \}
\]

Kahan (AVX) \{8 \parallel 4 \parallel 4 \parallel 4\} cy

Prediction \{8 \\backslash 8 \\backslash 12\} cy
ECM Model IvyBridge-EP

Model
Naïve (AVX): \{4 \parallel 4 \mid 4 \mid 4 \mid 7\} \text{cy} \quad \{4 \\parallel 8 \\parallel 12 \\parallel 19\} \text{cy}
Kahan (scalar): \{32 \parallel 8 \mid 4 \mid 4 \mid 7\} \text{cy} \quad \{32 \\parallel 32 \\parallel 32 \\parallel 32\} \text{cy}
Kahan (AVX): \{8 \parallel 4 \mid 4 \mid 4 \mid 7\} \text{cy} \quad \{8 \\parallel 8 \\parallel 12 \\parallel 19\} \text{cy}

Measurement
Naïve (AVX): 4.1 \parallel 8.7 \parallel 13.0 \parallel 24.9 \text{cy}
Kahan (scalar): 32.5 \parallel 32.4 \parallel 3248 \parallel 37.9 \text{cy}
Kahan (AVX): 8.4 \parallel 10.2 \parallel 13.7 \parallel 23.8 \text{cy}
Multicore scaling in the ECM model

Identify relevant **bandwidth bottlenecks**
- L3 cache
- Memory interface

**Scale** single-thread performance until **first bottleneck is hit**:

\[ P(t) = \min(tP_0, P_{roof}), \text{ with } P_{roof} = \min(P_{max}, l b_S) \]

Example: Scalable L3 on Sandy Bridge
ECM Model IvyBridge-EP

Model
Naïve (AVX): \{4 \parallel 4 \parallel 4 \parallel 4 \parallel 7\}cy \quad \{4 \parallel 8 \parallel 12 \parallel 19\}cy
Kahan (scalar): \{32 \parallel 8 \parallel 4 \parallel 4 \parallel 7\}cy \quad \{32 \parallel 32 \parallel 32 \parallel 32\}cy
Kahan (AVX): \{8 \parallel 4 \parallel 4 \parallel 4 \parallel 7\}cy \quad \{8 \parallel 8 \parallel 12 \parallel 19\}cy

Measurement
Naïve (AVX): 4.1 \parallel 8.7 \parallel 13.0 \parallel 24.9 cy
Kahan (scalar): 32.5 \parallel 32.4 \parallel 32.48 \parallel 37.9 cy
Kahan (AVX): 8.4 \parallel 10.2 \parallel 13.7 \parallel 23.8 cy
ECM prediction vs. measurements for \( A(:) = B(:) + C(:) \times D(:), \) no overlap

Model: Scales until saturation sets in

Saturation point (# cores) well predicted

Measurement: scaling not perfect

Caveat: This is specific for this architecture and this benchmark!

Check: Use “overlappable” kernel code
In-core execution is dominated by divide operation (44 cycles with AVX, 22 scalar)

→ Almost perfect agreement with ECM model

Parallelism “heals” bad single-core performance … just barely!
Case Study: Simplest code for the summation of the elements of a vector (single precision)

```c
float sum = 0.0;

for (int j=0; j<size; j++){
    sum += data[j];
}
```

Instruction code:

<table>
<thead>
<tr>
<th>Instruction address</th>
<th>Opcodes</th>
<th>Assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>401d08:</td>
<td>f3 0f 58 04 82</td>
<td>addss xmm0,[rdx + rax * 4]</td>
</tr>
<tr>
<td>401d0d:</td>
<td>48 83 c0 01</td>
<td>add rax,1</td>
</tr>
<tr>
<td>401d11:</td>
<td>39 c7</td>
<td>cmp edi,eax</td>
</tr>
<tr>
<td>401d13:</td>
<td>77 f3</td>
<td>ja 401d08</td>
</tr>
</tbody>
</table>

To get object code use `objdump -d` on object file or executable or compile with `-S`
Summation code (single precision): Optimizations

1:
addss xmm0, [rsi + rax * 4]
add rax, 1
cmp eax, edi
js 1b

1:
addss xmm0, [rsi + rax * 4]
add rax, 4
cmp eax, edi
js 1b

Unrolling with sub-sums to break up register dependency

3 cycles add pipeline latency

SSE SIMD vectorization
SIMD processing – single-threaded

**SIMD** influences instruction execution in the core – other bottlenecks stay the same!

- **Full benefit in L1 cache**
- **Data transfers are overlapped with execution**

**Execution Cache Memory**

- **Registers**
  - 48 cycles
  - 16 cycles
  - 4 cycles

- **L1D**
  - 32 byte LD & 16 byte ST
  - 2 cycles

- **L2**
  - 32 byte/cycle
  - 2 cycles

- **L3**
  - 15.6 byte/cycle
  - 4 cycles

- **Memory**

**Per-cacheline cycle counts**

- **Execution**
  - 48
  - 16
  - 4

**Per-cycle transfer widths**

- **Peak**
  - 15000
  - 12500
  - 10000
  - 7500
  - 5000
  - 2500
  - 0

- **Mflops/s**

- **L1**
  - 8cy
  - 16cy

- **L3**
  - 16cy

- **MEM**
  - 24cy

- **Some penalty for SIMD (12 cy predicted)**
And with AVX?

With preloading:
AVX down to less than 7 cycles (8309 MFlops/s)
SIMD processing – Full chip (all cores)
Influence of SMT

Bandwidth saturation is the primary performance limitation on the chip level!

Full scaling using SMT due to bubbles in pipeline

Conclusion: If the code saturates the bottleneck, all variants are acceptable!

All variants saturate the memory bandwidth

8 threads on physical cores

16 threads using SMT

Conclusion:
If the code saturates the bottleneck, all variants are acceptable!
The ECM model is a simple analysis tool to get insight into:

- Runtime contributions
- Bottleneck identification
- Runtime overlap

It can predict single core performance for any memory hierarchy level and get an estimate of multicore chip scalability.

**ECM correctly describes several effects**

- Saturation for memory-bound loops
- Diminishing returns of in-core optimizations for far-away data

Simple models work best. Do not try to complicate things unless it is really necessary!
CASE STUDY: HPCCG

Performance analysis on:

- Intel IvyBridge-EP@2.2GHz
- Intel Xeon Phi@1.05GHz
for(int k=1; k<max_iter && normr > tolerance; k++ )
{
    oldrtrans = rtrans;
    ddot (nrow, r, r, &rtrans, t4);
    double beta = rtrans/oldrtrans;
    waxpby (nrow, 1.0, r, beta, p, p);
    normr = sqrt(rtrans);
    HPC_sparsemv(A, p, Ap);
    double alpha = 0.0;
    ddot(nrow, p, Ap, &alpha, t4);
    alpha = rtrans/alpha;
    waxpby(nrow, 1.0, r, -alpha, Ap, r);
    waxpby(nrow, 1.0, x, alpha, p, x);
    niters = k;
}
Components of HPCCG 1

ddot:

```cpp
#pragma omp for reduction (+:result)
for (int i=0; i<n; i++) {
    result += x[i] * y[i];
}
```

waxpby:

```cpp
#pragma omp for
for (int i=0; i<n; i++) {
    w[i] = alpha * x[i] + beta * y[i];
}
```

2 Flops
2 * 8b L = 16b
2.2GHz/2c * 16 Flops =
17.6 GFlops/s or
140GB/s L1 or 46GB/s L2

3 Flops
2 * 8b L + 1 * 8b S = 24b
2.2GHz/4c * 24flops =
13.2 GFlops/s or
106GB/s L1 or 47GB/s L2
Sparse matrix-vector multiply (spMVM)

- Key ingredient in some matrix diagonalization algorithms
  - Lanczos, Davidson, Jacobi-Davidson
- **Store only** $N_{nz}$ **nonzero elements** of matrix and RHS, LHS vectors with $N_r$ (number of matrix rows) entries
- “Sparse”: $N_{nz} \sim N_r$

\[ \begin{bmatrix} \text{v} \\ \end{bmatrix} = \begin{bmatrix} \text{v} \\ \end{bmatrix} + \begin{bmatrix} \text{v} \\ \end{bmatrix} \cdot N_r \]

General case: some indirect addressing required!
CRS matrix storage scheme

- **val[]** stores all the nonzeros (length $N_{nz}$)
- **col_idx[]** stores the column index of each nonzero (length $N_{nz}$)
- **row_ptr[]** stores the starting index of each new row in **val[]** (length: $N_r$)
CRS (Compressed Row Storage) – data format

Format creation
1. Store values and column indices of all non-zero elements row-wise
2. Store starting indices of each column (\texttt{rpt})

Data arrays
- \texttt{double val[]} 
- \texttt{unsigned int col[]} 
- \texttt{unsigned int rpt[]}
Components of HPCCG 2

```c
#pragma omp for
for (int i=0; i< nrow; i++) {
    double sum = 0.0;
    double* cur_vals = vals_in_row[i];
    int* cur_inds = inds_in_row[i];
    int cur_nnz = nnz_in_row[i];
    for (int j=0; j< cur_nnz; j++) {
        sum += cur_vals[j]*x[cur_inds[j]];
    }
    y[i] = sum;
}
```

2 Flops
1 * 4b L + 2 * 8b L = 20b
2.2GHz/2c * 16 Flops =
17.6 GFlops/s or
140GB/s L1 or 46GB/s L2
First Step: Runtime Profile (300^3)

Intel IvyBridge-EP (2.2GHz, 10 cores/chip)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Serial</th>
<th>Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddot</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>waxby</td>
<td>12%</td>
<td>16%</td>
</tr>
<tr>
<td>spmv</td>
<td>83%</td>
<td>79%</td>
</tr>
</tbody>
</table>

Intel Xeon Phi (1.05GHz, 60 cores/chip)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddot</td>
<td>3%</td>
</tr>
<tr>
<td>waxby</td>
<td>8%</td>
</tr>
<tr>
<td>spmv</td>
<td>89%</td>
</tr>
</tbody>
</table>
Scaling behavior inside socket (IvyBridge-EP)

HPM measurement with LIKWID instrumentation on socket level

<table>
<thead>
<tr>
<th>Routine</th>
<th>Time [s]</th>
<th>Memory Bandwidth [MB/s]</th>
<th>Data Volume [GB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>waxby 1</td>
<td>2,33</td>
<td>40464</td>
<td>93</td>
</tr>
<tr>
<td>waxby 2</td>
<td>2,37</td>
<td>39919</td>
<td>94</td>
</tr>
<tr>
<td>waxby 3</td>
<td>2,4</td>
<td>40545</td>
<td>96</td>
</tr>
<tr>
<td>ddot 1</td>
<td>0,72</td>
<td>46886</td>
<td>34</td>
</tr>
<tr>
<td>ddot 2</td>
<td>1,4</td>
<td>46444</td>
<td>64</td>
</tr>
<tr>
<td>spmv</td>
<td>33,84</td>
<td>45964</td>
<td>1555</td>
</tr>
</tbody>
</table>
Scaling to full node \((180^3)\)

### Performance [GFlops/s]

<table>
<thead>
<tr>
<th>Routine</th>
<th>Socket</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddot</td>
<td>6726</td>
<td>14547</td>
</tr>
<tr>
<td>waxby</td>
<td>3642</td>
<td>6123</td>
</tr>
<tr>
<td>spmv</td>
<td>6374</td>
<td>6320</td>
</tr>
<tr>
<td>Total</td>
<td>5973</td>
<td>6531</td>
</tr>
</tbody>
</table>

### Memory Bandwidth measured [GB/s]

<table>
<thead>
<tr>
<th>Routine</th>
<th>Socket 1</th>
<th>Socket 2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddot</td>
<td>44020</td>
<td>47342</td>
<td>91362</td>
</tr>
<tr>
<td>waxby</td>
<td>39795</td>
<td>28424</td>
<td>68219</td>
</tr>
<tr>
<td>spmv</td>
<td>43109</td>
<td>2863</td>
<td>45972</td>
</tr>
</tbody>
</table>

Pattern: Bad ccNUMA page placement
Matrix data was not placed. **Solution:** Add first touch initialization.

```c
#pragma omp parallel for
for (int i=0; i< local_nrow; i++){
    for (int j=0; j< 27; j++) {
        curvalptr[i*27 + j] = 0.0;
        curindptr[i*27 + j] = 0;
    }
}
```

**Node performance: spmv 11692, total 10912**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Socket 1</th>
<th>Socket 2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddot</td>
<td>46406</td>
<td>48193</td>
<td>94599</td>
</tr>
<tr>
<td>waxby</td>
<td>37113</td>
<td>24904</td>
<td>62017</td>
</tr>
<tr>
<td>spmv</td>
<td>45822</td>
<td>40935</td>
<td>86757</td>
</tr>
</tbody>
</table>
Scaling behavior Intel Xeon Phi

Code is instruction throughput limited
Pattern: Expensive Instructions

134804 MB/s
131803 MB/s
Format creation

1. Shift nonzeros in each row to the left
2. Combine chunkHeight (multiple of vector length, here: 8) rows to one chunk
3. Pad all rows in chunk to the same length
4. Store matrix chunk by chunk and jagged-diagonal-wise within chunk

Data arrays

double val[]
unsigned int col[]
unsigned int chunkStart[]
Optimized spmv data structure on Xeon Phi

Pattern: Bandwidth saturation
EMPLOYING THE ECM MODEL ON STENCIL KERNELS
2D Jacobi Stencil: Layer condition

![Graph showing performance vs. leading dimension (N_i) for different memory bandwidths (B_C, B^L3, B^mem)].

- **LC in L2**: Performance drops significantly due to limited memory access.
- **LC in L3**: Performance improves as memory bandwidth is increased to 40 B/LUP.
- **No LC**: Performance remains relatively stable with additional bandwidths.

Performance [MLUP/s]

Leading dimension (N_i)

- B^mem = 24 B/LUP
- B_C = 24 B/LUP
- B_C^L3 = 40 B/LUP
J2D multicore chip scaling
uxx stencil from earthquake propagation code

for(int k=2; k<=N-1; k++){
    for (int j=2; j<=N-1; j++){
        for (int i=2; i<=N-1; i++){
            d = 0.25*(d1[k][j][i] + d1[k][j-1][i] + d1[k-1][j][i] + d1[k-1][j-1][i]);
            u1[k][j][i] = u1[k][j][i] + (dth/d) *( c1*(xx[k][j][i]-xx[k][j][i-1])
                                                + c2*(xx[k][j][i+1]-xx[k][j][i-2])
                                                + c1*(xy[k][j][i]-xy[k][j-1][i])
                                                + c2*(xy[k][j+1][i]-xy[k][j-2][i])
                                                + c1*(xz[k][j][i]-xz[k-1][j][i])
                                                + c2*(xz[k+1][j][i]-xz[k-2][j][i]));
        }
    }
}

\textbf{vdivpd}: 42 cycles throughput in double precision (SNB)

What about single precision?
uxx kernel ECM model

Employing the Intel IACA tool for L1 throughput estimate.

<table>
<thead>
<tr>
<th>Version</th>
<th>ECM model</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>{84 , 38 , 20 , 20 , 26} \text{cy}</td>
<td>{84 , 84 , 84 , 104} \text{cy}</td>
</tr>
<tr>
<td>SP</td>
<td>{45 , 38 , 20 , 20 , 26} \text{cy}</td>
<td>{45 , 58 , 78 , 104} \text{cy}</td>
</tr>
<tr>
<td>DP noDIV</td>
<td>{41 , 38 , 20 , 20 , 26} \text{cy}</td>
<td>{41 , 58 , 78 , 104} \text{cy}</td>
</tr>
</tbody>
</table>

Prediction for in Memory data set:
1. SP is twice as fast as DP
2. All variants saturate at 4 cores
3. The presence of the DIV in DP makes no difference
Comparison model vs. measurement

(a) Sandy Bridge
(b) Ivy Bridge
uxx kernel: Optimization opportunities

- ECM model allows to predict upper limit for benefits from temporal blocking for the L3 cache:
  - Removes L3-MEM transfer time of 26cy
  - 24% speedup in DP (single core)
  - 33% speedup in SP (single core)

- Next bottleneck is the divide (DP) and L3 transfers (SP).
- True benefit: Both are core-local and therefore scalable.

- Expected performance in DP on chip level 2000 MLUP/s instead of 800 MLUPS/s (even with DIV)