Node-Level Performance Engineering

Georg Hager, Jan Eitzinger, Gerhard Wellein
Erlangen Regional Computing Center (RRZE)
and Department of Computer Science
University of Erlangen-Nuremberg

SC17 full-day tutorial
November 12, 2017
Denver, CO, USA

For final slides and example code see: https://goo.gl/ixCesm

☆: slide updated from USB version
<table>
<thead>
<tr>
<th>Time</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>08:30</td>
<td><strong>Preliminaries</strong></td>
</tr>
<tr>
<td>10:00</td>
<td><strong>Introduction to multicore architecture</strong></td>
</tr>
<tr>
<td></td>
<td>- Threads, cores, SIMD, caches, chips, sockets, ccNUMA</td>
</tr>
<tr>
<td>10:30</td>
<td><strong>Multicore tools</strong></td>
</tr>
<tr>
<td>12:00</td>
<td><strong>Microbenchmarking for architectural exploration</strong></td>
</tr>
<tr>
<td></td>
<td>- Streaming benchmarks</td>
</tr>
<tr>
<td></td>
<td>- Hardware bottlenecks</td>
</tr>
<tr>
<td>13:30</td>
<td><strong>Node-level performance modeling (part I)</strong></td>
</tr>
<tr>
<td></td>
<td>- The Roofline Model and dense MVM</td>
</tr>
<tr>
<td>13:30</td>
<td><strong>Lunch break</strong></td>
</tr>
<tr>
<td>15:00</td>
<td><strong>Node-level performance modeling (part II)</strong></td>
</tr>
<tr>
<td></td>
<td>- Case studies: Jacobi solver, Sparse MVM</td>
</tr>
<tr>
<td>15:30</td>
<td><strong>Optimal resource utilization</strong></td>
</tr>
<tr>
<td></td>
<td>- SIMD parallelism</td>
</tr>
<tr>
<td></td>
<td>- ccNUMA</td>
</tr>
<tr>
<td></td>
<td>- OpenMP synchronization and multicores</td>
</tr>
<tr>
<td>17:00</td>
<td><strong>Pattern-driven performance engineering</strong></td>
</tr>
</tbody>
</table>
A conversation

From a student seminar on “Efficient programming of modern multi- and manycore processors”

Student: I have implemented this algorithm on the GPGPU, and it solves a system with 26546 unknowns in 0.12 seconds, so it is really fast.

Me: What makes you think that 0.12 seconds is fast?

Student: It is fast because my baseline C++ code on the CPU is about 20 times slower.
Prelude:
Scalability 4 the win!
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

3D Stencil Update
("Jacobi")

Speed-Up

- Version 1
- Version 2

Prepared for the highly parallel era!

-03 -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) = \( b \times (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1)) \)
    enddo; enddo
  enddo
!$OMP END PARALLEL DO
```

Single core/socket efficiency is key issue!

Upper limit from simple performance model:
35 GB/s & 24 Byte/update
Questions to ask in high performance computing

- Do I understand the performance behavior of my code?
  - Does the performance match a model I have made?

- What is the optimal performance for my code on a given machine?
  - High Performance Computing == Computing at the bottleneck

- Can I change my code so that the “optimal performance” gets higher?
  - Circumventing/ameliorating the impact of the bottleneck

- My model does not work – what’s wrong?
  - This is the good case, because you learn something
  - Performance monitoring / microbenchmarking may help clear up the situation
How model-building works: Physics

Newtonian mechanics

\[ \vec{F} = m \vec{a} \]

Nonrelativistic quantum mechanics

\[ i \hbar \frac{\partial}{\partial t} \psi(\vec{r}, t) = \hat{H} \psi(\vec{r}, t) \]

Fails @ small scales!

If a model fails, we learn something!

Relativistic quantum field theory

\[ U(1)_Y \otimes SU(2)_L \otimes SU(3)_c \]

(c) RRZE 2017

Node-Level Performance Engineering
Introduction: Modern node architecture

A glance at basic core features:
  - pipelining, superscalarity, SMT
  - Caches and data transfers through the memory hierarchy
  - Accelerators
  - Bottlenecks & hardware-software interaction
Multi-core today: Intel Xeon 2600v4 (2016)

- Xeon E5-2600v4 “Broadwell EP”:
  Up to 22 cores running at 2+ GHz (+ “Turbo Mode”: 3.5+ GHz)

- Simultaneous Multithreading → reports as 44-way chip

- 7.2 Billion Transistors / 14 nm

- Die size: 456 mm²

Optional: “Cluster on Die” (CoD) mode

2017: Skylake architecture
- Mesh instead of ring interconnect
- Sub-NUMA clustering
- Up to 28 cores
- 2.5 → 3.8 GHz (top bin)
General-purpose cache based microprocessor core

- Implements “Stored Program Computer” concept (Turing 1936)
- Similar designs on all modern systems
- (Still) multiple potential bottlenecks
- The clock cycle is the “heartbeat” of the core

Modern CPU core

(c) RRZE 2017
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 – sufficient # of independent instructions required
  - Requires complex instruction scheduling by compiler/hardware
    - software-pipelining / out-of-order execution

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

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

Wind-up/-down phases: Empty pipeline stages

(c) RRZE 2017
Node-Level Performance Engineering
Pipelining: The Instruction pipeline

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

**Diagram:**

- 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)

(c) RRZE 2017
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) “logical” cores → multiple threads/processes run concurrently

SMT principle (2-way example):
SMT impact

- **Possible benefit:** Better pipeline throughput
  - Filling otherwise unused pipelines
  - Filling pipeline bubbles with other thread’s executing instructions:

  **Thread 0:**
  ```
  do i=1,N
  a(i) = a(i-1)*c
  enddo
  ```

  **Thread 1:**
  ```
  do i=1,N
  b(i) = s*b(i-2)+d
  enddo
  ```

  **Dependency → pipeline stalls until previous MULT is over**

  - **Beware:** Executing it all in a single thread (if possible) may achieve the same goal without SMT:
  ```
  do i=1,N
  a(i) = a(i-1)*c
  b(i) = s*b(i-2)+d
  enddo
  ```

  - **Caveat:** SMT adds another layer of topology (inside the physical core)
  - **Caveat:** SMT threads share all caches!
Core details: SIMD processing

- Single Instruction Multiple Data (SIMD) operations allow the concurrent execution of the same operation on “wide” registers

- x86 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands
  - AXV512: you get it.

- Adding two registers holding double precision floating point operands

Scalar execution:
\[ \text{ADD} \left[ R0, R1 \right] \rightarrow R2 \]

SIMD execution:
\[ \text{V}_{64}\text{ADD} \left[ R0, R1 \right] \rightarrow R2 \]
**SIMD processing – Basics**

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

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

- **“Loop unrolling”**

```c
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

---

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]

---

**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

This should not be done by hand!
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" does not happen, tell 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) {...}
```
There is no single driving force for single core performance!

Maximum floating point (FP) performance:

\[ P_{\text{core}} = n_{\text{super}}^{\text{FP}} \cdot n_{\text{FMA}} \cdot n_{\text{SIMD}} \cdot f \]

<table>
<thead>
<tr>
<th>Typical representatives</th>
<th>( n_{\text{super}}^{\text{FP}} ) inst./cy</th>
<th>( n_{\text{FMA}} )</th>
<th>( n_{\text{SIMD}} ) ops/instr.</th>
<th>Code</th>
<th>( f ) [GHz]</th>
<th>( P_{\text{core}} ) [GF/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>Q1/2009</td>
<td>2.93</td>
<td>11.7</td>
</tr>
<tr>
<td>Westmere</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>Q1/2010</td>
<td>2.66</td>
<td>10.6</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>Q1/2012</td>
<td>2.7</td>
<td>21.6</td>
</tr>
<tr>
<td>Ivy Bridge</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>Q3/2013</td>
<td>2.2</td>
<td>17.6</td>
</tr>
<tr>
<td>Haswell</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>Q3/2014</td>
<td>2.3</td>
<td>36.8</td>
</tr>
<tr>
<td>Broadwell</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>Q1/2016</td>
<td>2.2</td>
<td>35.2</td>
</tr>
<tr>
<td>IBM POWER8</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>Q2/2014</td>
<td>2.93</td>
<td>23.4</td>
</tr>
</tbody>
</table>

(c) RRZE 2017

Node-Level Performance Engineering
Registers and caches: Data transfers in a memory hierarchy

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 the data in a cache level → CL transfer required

- Example: Array copy \( A(:) = C(:) \)
Yesterday (2006): UMA

Uniform Memory Architecture (UMA)
Flat memory; symmetric MPs
But: system “anisotropy”

Today: ccNUMA

Cache-coherent Non-Uniform Memory Architecture (ccNUMA)
Two or more NUMA domains per node
ccNUMA provides scalable bandwidth but: Where does my data finally end up?
Interlude:
A glance at current accelerator technology

NVidia “Pascal” GP100
vs.
Intel Xeon Phi “Knights Landing”
Architecture

- 15.3 B Transistors
- ~ 1.4 GHz clock speed
- Up to 60 “SM” units
  - 64 SP “cores” each
  - 32 DP “cores” each
  - 2:1 SP:DP performance
- 5.7 TFlop/s DP peak
- 4 MB L2 Cache
- 4096-bit HBM2
- MemBW ~ 732 GB/s (theoretical)
- MemBW ~ 510 GB/s (measured)
Intel Xeon Phi “Knights Landing” block diagram

**Architecture**
- 8 B Transistors
- Up to 1.5 GHz clock speed
- Up to 36x2 cores (2D mesh)
  - 2x 512-bit SIMD units (VPU) each
  - 4-way SMT
- 3.5 TFlop/s DP peak (SP 2x)
- 36 MiB L2 Cache
- 16 GiB MCDRAM
  - MemBW ~ 470 GB/s (measured)
- Large DDR4 main memory
  - MemBW ~ 90 GB/s (measured)
Trading single thread performance for parallelism: *GPGPUs vs. CPUs*

**GPU vs. CPU**

light speed estimate  
(per device)

- MemBW: ~ 5-10x
- Peak: ~ 5-10x

### Comparison Table

<table>
<thead>
<tr>
<th></th>
<th>2x Intel Xeon E5-2697v4 “Broadwell”</th>
<th>Intel Xeon Phi 7250 “Knights Landing”</th>
<th>NVidia Tesla P100 “Pascal”</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cores@Clock</strong></td>
<td>2 x 18 @ ≥2.3 GHz</td>
<td>68 @ 1.4 GHz</td>
<td>56 SMs @ ~1.3 GHz</td>
</tr>
<tr>
<td><strong>SP Performance/core</strong></td>
<td>≥73.6 GFlop/s</td>
<td>89.6 GFlop/s</td>
<td>~166 GFlop/s</td>
</tr>
<tr>
<td><strong>Threads@STREAM</strong></td>
<td>~8</td>
<td>~40</td>
<td>&gt; 10000</td>
</tr>
<tr>
<td><strong>SP peak</strong></td>
<td>≥2.6 TFlop/s</td>
<td>6.1 TFlop/s</td>
<td>~9.3 TFlop/s</td>
</tr>
<tr>
<td><strong>Stream BW (meas.)</strong></td>
<td>2 x 62.5 GB/s</td>
<td>450 GB/s (HBM)</td>
<td>510 GB/s</td>
</tr>
<tr>
<td><strong>Transistors / TDP</strong></td>
<td>~2x7 Billion / 2x145 W</td>
<td>8 Billion / 215W</td>
<td>14 Billion/300W</td>
</tr>
</tbody>
</table>

(c) RRZE 2017

Node-Level Performance Engineering
Node topology and programming models
Parallelism in a modern compute node

- Parallel and shared resources (potential bottlenecks!) within a shared-memory node

**Parallel resources:**
- Execution/SIMD units
- Cores
- Inner cache levels
- Sockets / ccNUMA domains
- Multiple accelerators

**Shared resources:**
- Outer cache level per socket
- Memory bus per socket
- Intersocket link
- PCIe bus(es)
- Other I/O resources

How does your application react to all of those details?

(c) RRZE 2017
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

![Diagram showing communication network and processor placements](chart.png)
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
Conclusions about architecture

- 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)
  - Superscalarity (CPU: 2-6)

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

- **Performance of programming models is sensitive to architecture**
  - Topology/affinity influences overheads
  - Standards do not contain (many) topology-aware features
  - Apart from overheads, performance features are largely independent of the programming model
Multicore Performance and Tools
Tools for Node-level Performance Engineering

- **Gather Node Information**
  - hwloc, **likwid-topology**, likwid-powermeter

- **Affinity control** and data placement
  - OpenMP and MPI runtime environments, hwloc, numactl, **likwid-pin**

- **Runtime Profiling**
  - Compilers, gprof, HPC Toolkit, …

- **Performance Profilers**
  - Intel Vtune™, **likwid-perfctr**, PAPI based tools, Linux perf, …

- **Microbenchmarking**
  - STREAM, **likwid-bench**, Imbench
LIKWID performance tools

LIKWID tool suite:

Like I Knew What I’m Doing

Open source tool collection (developed at RRZE):
https://github.com/RRZE-HPC/likwid

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-powermeter`: Measure energy consumption
  - `likwid-pin`: Pin threaded application without touching code
  - `likwid-perfctr`: Measure performance counters
  - `likwid-bench`: Microbenchmarking tool and environment
  - … some more

(c) RRZE 2017
Output of `likwid-topology -g` on one node of Intel Haswell-EP

---

**CPU name:** Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz  
**CPU type:** Intel Xeon Haswell EN/EP/EX processor  
**CPU stepping:** 2

---

### Hardware Thread Topology

---

<table>
<thead>
<tr>
<th>HWThread</th>
<th>Thread</th>
<th>Core</th>
<th>Socket</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>

---

**Socket 0:** ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 7 35 8 36 9 37 10 38 11 39 12 40 13 41 )  
**Socket 1:** ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 21 49 22 50 23 51 24 52 25 53 26 54 27 55 )

---

### Cache Topology

---

**Level 1:**  
**Size:** 32 kB  
**Cache groups:** ( 0 28 ) ( 1 29 ) ( 2 30 ) ( 3 31 ) ( 4 32 ) ( 5 33 ) ( 6 34 ) ( 7 35 ) ( 8 36 ) ( 9 37 ) ( 10 38 ) ( 11 39 ) ( 12 40 ) ( 13 41 ) ( 14 42 ) ( 15 43 ) ( 16 44 ) ( 17 45 ) ( 18 46 ) ( 19 47 ) ( 20 48 ) ( 21 49 ) ( 22 50 ) ( 23 51 ) ( 24 52 ) ( 25 53 ) ( 26 54 ) ( 27 55 )

---

**Level 2:**  
**Size:** 256 kB  
**Cache groups:** ( 0 28 ) ( 1 29 ) ( 2 30 ) ( 3 31 ) ( 4 32 ) ( 5 33 ) ( 6 34 ) ( 7 35 ) ( 8 36 ) ( 9 37 ) ( 10 38 ) ( 11 39 ) ( 12 40 ) ( 13 41 ) ( 14 42 ) ( 15 43 ) ( 16 44 ) ( 17 45 ) ( 18 46 ) ( 19 47 ) ( 20 48 ) ( 21 49 ) ( 22 50 ) ( 23 51 ) ( 24 52 ) ( 25 53 ) ( 26 54 ) ( 27 55 )

---

**Level 3:**  
**Size:** 17 MB  
**Cache groups:** ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 ) ( 7 35 8 36 9 37 10 38 11 39 12 40 13 41 ) ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 ) ( 21 49 22 50 23 51 24 52 25 53 26 54 27 55 )

---

(c) RRZE 2017

Node-Level Performance Engineering
NUMA Topology

<table>
<thead>
<tr>
<th>Domain:</th>
<th>Processors</th>
<th>Distances</th>
<th>Free memory:</th>
<th>Total memory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 28 1 29 2 30 3 31 4 32 5 33 6 34</td>
<td>10 21 31 31</td>
<td>13292.9 MB</td>
<td>15941.7 MB</td>
</tr>
<tr>
<td>1</td>
<td>7 35 8 36 9 37 10 38 11 39 12 40 13 41</td>
<td>21 10 31 31</td>
<td>13514 MB</td>
<td>16126.4 MB</td>
</tr>
<tr>
<td>2</td>
<td>14 42 15 43 16 44 17 45 18 46 19 47 20 48</td>
<td>31 31 10 21</td>
<td>15025.6 MB</td>
<td>16126.4 MB</td>
</tr>
<tr>
<td>3</td>
<td>21 49 22 50 23 51 24 52 25 53 26 54 27 55</td>
<td>31 31 21 10</td>
<td>15488.9 MB</td>
<td>16126 MB</td>
</tr>
</tbody>
</table>
Output of likwid-topology continued

-----------------------------------------------------------------------------------------------------------------------------
Socket 0:
|  0 28 |  1 29 |  2 30 |  3 31 |  4 32 |  5 33 |  6 34 |  7 35 |  8 36 |  9 37 | 10 38 | 11 39 | 12 40 | 13 41 |
|       |       |       |       |       | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |       |
+---------------------------------------------------------------------------------------------------------------
17MB

Socket 1:
|  14 42 |  15 43 |  16 44 |  17 45 |  18 46 |  19 47 |  20 48 |  21 49 |  22 50 |  23 51 |  24 52 |  25 53 |  26 54 |  27 55 |
|       |       |       |       | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
| +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +     | +      |
+---------------------------------------------------------------------------------------------------------------
17MB

Cluster on die mode and SMT enabled!
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

There are several reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention
More thread/Process-core affinity ("pinning") options

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


- 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 (see OpenMP tutorial)

- Affinity awareness in MPI libraries
  - SGI MPT
  - 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 be used as a superior replacement for taskset
- Supports logical core numbering within a node

**Usage examples:**

- `likwid-pin -c 0-3,4,6 ./myApp parameters`
- `likwid-pin -c S0:0-7 ./myApp parameters`
- `likwid-pin -c N:0-15 ./myApp parameters`

- `OMP_NUM_THREADS` is set by the tool if not set explicitly
LIKWID terminology

Thread group syntax

- The OS numbers all processors (hardware threads) on a node
- The numbering is enforced at boot time by the BIOS and may have nothing to do with topological entities
- LIKWID concept: **thread group** consisting of HW threads sharing a topological entity (e.g., socket, shared cache,…)
- A thread group is defined by a single **character + index**

**Example:**
```
likwid-pin -c S1:0-3,6,7 ./a.out
```

- Group expression chaining with @:
```
likwid-pin -c S0:0-3@S1:0-3 ./a.out
```

- Alternative expression based syntax:
```
likwid-pin -c E:S0:4:2:4 ./a.out
```

  - `E:<thread domain>:<num threads>:<chunk size>:<stride>`

- Expression syntax is convenient for Xeon Phi:
```
likwid-pin -c E:N:120:2:4 ./a.out
```
Likwid

Currently available thread domains

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

Default if –c is not specified!
Running the STREAM benchmark with likwid-pin:

$ likwid-pin -c S0:0-3 ./stream

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

Array size = 20000000
Offset = 32
The total memory requirement is 457 MB
You are running each test 10 times

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

[pthread wrapper] MAIN -> 0
[pthread wrapper] PIN_MASK: 0->1 1->2 2->3
[pthread wrapper] SKIP MASK: 0x1
   threadid 140668624234240 -> SKIP
   threadid 140668598843264 -> core 1 - OK
   threadid 140668594644992 -> core 2 - OK
   threadid 140668590446720 -> core 3 - OK

[... rest of STREAM output omitted ...]
Intel KMP_AFFINITY environment variable

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

**modifier**
- \texttt{granularity=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

**KMP\_AFFINITY=verbose,none** to list machine topology map
Intel **KMP\_AFFINITY** examples

- **KMP\_AFFINITY=granularity=fine, compact**

  (c) Intel

  ![Diagram](image)

  Package means chip/socket

- **KMP\_AFFINITY=granularity=fine, scatter**

  (c) Intel

  ![Diagram](image)

  OpenMP\* global thread IDs

  Thread context

(c) RRZE 2017
Intel KMP_AFFINITY permute example

- **KMP_AFFINITY=granularity=fine,compact,1,0**

(c) Intel

- **KMP_AFFINITY=granularity=core,compact**

(c) Intel

Threads may float within core
**GNU GOMP_AFFINITY**

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

(c) Intel

- Always operates with **OS processor IDs**

Round robin oversubscription
Multicore performance tools:
Probing performance behavior

likwid-perfctr
Basic approach to performance analysis

1. Runtime profile / Call graph (gprof): Where are the hot spots?
2. Instrument hot spots (prepare for detailed measurement)
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)

Goal: Come up with educated guess about a performance-limiting motif (Performance Pattern)

(c) RRZE 2017

Node-Level Performance Engineering
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
  - 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 likwid-perfctr -a

- BRANCH: Branch prediction miss rate/ratio
- CACHE: Data cache miss rate/ratio
- CLOCK: Clock of cores
- DATA: Load to store ratio
- FLOPS_DP: Double Precision MFlops/s
- FLOPS_SP: Single Precision MFlops/s
- FLOPS_X87: X87 MFlops/s
- L2: L2 cache bandwidth in MBytes/s
- L2CACHE: L2 cache miss rate/ratio
- L3: L3 cache bandwidth in MBytes/s
- L3CACHE: L3 cache miss rate/ratio
- MEM: Main memory bandwidth in MBytes/s
- TLB: TLB miss rate/ratio
Example usage with preconfigured metric group (shortened)

$ likwid-perfctr -C N:0-3 -g FLOPS_DP ./stream.exe

CPU name: Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz
CPU type: Intel Xeon IvyBridge EN/EP/EX processor
CPU clock: 2.20 GHz

[... YOUR PROGRAM OUTPUT ...]

Group 1: FLOPS_DP

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</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>FIXC0</td>
<td>521332883</td>
<td>523904122</td>
<td>519696583</td>
<td>519193735</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>FIXC1</td>
<td>1379625927</td>
<td>1381900036</td>
<td>1378355460</td>
<td>1376447</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>FIXC2</td>
<td>1389460886</td>
<td>1393031508</td>
<td>1387504228</td>
<td>1385276</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE</td>
<td>PMC0</td>
<td>176216849</td>
<td>176176025</td>
<td>177432054</td>
<td>17636756</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE</td>
<td>PMC1</td>
<td>1554</td>
<td>599</td>
<td>72</td>
<td>27</td>
</tr>
<tr>
<td>SIMD_FP_256_PACKED_DOUBLE</td>
<td>PMC2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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 (RDTSC) [s]</td>
<td>0.6856</td>
<td>0.6856</td>
<td>0.6856</td>
<td>0.6856</td>
</tr>
<tr>
<td>Runtime unhalted [s]</td>
<td>0.6270</td>
<td>0.6281</td>
<td>0.6265</td>
<td>0.6256</td>
</tr>
<tr>
<td>Clock [MHz]</td>
<td>2184.6742</td>
<td>2182.6664</td>
<td>2185.7404</td>
<td>2186.2243</td>
</tr>
<tr>
<td>CPI</td>
<td>2.6463</td>
<td>2.6377</td>
<td>2.6522</td>
<td>2.6511</td>
</tr>
<tr>
<td>MFLOP/s</td>
<td>514.0890</td>
<td>513.9685</td>
<td>517.6320</td>
<td>514.5273</td>
</tr>
<tr>
<td>AVX MFLOP/s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Packed MUOPS/s</td>
<td>257.0434</td>
<td>256.9838</td>
<td>258.8160</td>
<td>257.2636</td>
</tr>
<tr>
<td>Scalar MUOPS/s</td>
<td>0.0023</td>
<td>0.0009</td>
<td>0.0001</td>
<td>3.938426e-05</td>
</tr>
</tbody>
</table>
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 region support, accumulation over multiple calls

Inclusive and overlapping regions allowed

```c
#include <likwid.h>
...
LIKWD_MARKER_INIT; // must be called from serial region
#pragma omp parallel
{
    LIKWD_MARKER_THREADINIT; // only reqd. if measuring multiple threads
}
...
LIKWD_MARKER_START("Compute");
...
LIKWD_MARKER_STOP("Compute");
...
LIKWD_MARKER_START("Postprocess");
...
LIKWD_MARKER_STOP("Postprocess");
...
LIKWD_MARKER_CLOSE; // must be called from serial region
```

- **Activate macros with** `-DLIKWD_PERFMON`
- **Run likwid-perfctr with** `-m` **option to activate markers**
Best practices for runtime counter analysis

Things to look at (in roughly this order)

- Excess work
- Load balance (flops, instructions, BW)
- In-socket memory BW saturation
- Flop/s, loads and stores per flop metrics
- SIMD vectorization
- CPI metric
- # of instructions, branches, mispredicted branches

Caveats

- Load imbalance may not show in CPI or # of instructions
  - Spin loops in OpenMP barriers/MPI blocking calls
  - Looking at “top” or the Windows Task Manager does not tell you anything useful
- In-socket performance saturation may have various reasons
- Cache miss metrics are sometimes misleading
Measuring energy consumption with LIKWID
Measuring energy consumption
likwid-powermeter and likwid-perfctr -g ENERGY

- Implements Intel RAPL interface (Sandy Bridge)
- RAPL = “Running average power limit”

-----------------------------------------------------------------------
CPU name: Intel Core SandyBridge processor
CPU clock: 3.49 GHz
-----------------------------------------------------------------------
Base clock: 3500.00 MHz
Minimal clock: 1600.00 MHz
Turbo Boost Steps:
C1 3900.00 MHz
C2 3800.00 MHz
C3 3700.00 MHz
C4 3600.00 MHz
-----------------------------------------------------------------------
Thermal Spec Power: 95 Watts
Minimum Power: 20 Watts
Maximum Power: 95 Watts
Maximum Time Window: 0.15625 micro sec
-----------------------------------------------------------------------
Example:
A medical image reconstruction code on Sandy Bridge

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8 cores, plain C</td>
<td>90.43</td>
<td>90</td>
<td>8110</td>
</tr>
<tr>
<td>8 cores, SSE</td>
<td>29.63</td>
<td>93</td>
<td>2750</td>
</tr>
<tr>
<td>8 cores (SMT), SSE</td>
<td>22.61</td>
<td>102</td>
<td>2300</td>
</tr>
<tr>
<td>8 cores (SMT), AVX</td>
<td>18.42</td>
<td>111</td>
<td>2040</td>
</tr>
</tbody>
</table>

Sandy Bridge EP (8 cores, 2.7 GHz base freq.)

Faster code \(\rightarrow\) less energy
Microbenchmarking for architectural exploration (and more)

Probing of the memory hierarchy
Saturation effects in cache and memory
Latency and bandwidth in modern computer environments

Avoiding slow data paths is the key to most performance optimizations!
## Intel Xeon E5 multicore processors

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Shorthand</td>
<td>SNB</td>
<td>IVB</td>
<td>HSW</td>
</tr>
<tr>
<td>Xeon Model</td>
<td>E5-2680</td>
<td>E5-2690 v2</td>
<td>E5-2695 v3</td>
</tr>
<tr>
<td>Year</td>
<td>03/2012</td>
<td>09/2013</td>
<td>09/2014</td>
</tr>
<tr>
<td>Clock speed (fixed)</td>
<td>2.7 GHz</td>
<td>2.2 GHz</td>
<td>2.3 GHz</td>
</tr>
<tr>
<td>Cores/Threads</td>
<td>8/16</td>
<td>10/20</td>
<td>14/28</td>
</tr>
</tbody>
</table>

### Load/Store throughput per cycle

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX(2)</td>
<td>1 LD &amp; 1/2 ST</td>
<td>1 LD &amp; 1/2 ST</td>
<td>2 LD &amp; 1 ST</td>
</tr>
<tr>
<td>SSE/scalar</td>
<td>2 LD</td>
<td></td>
<td>1 LD &amp; 1 ST</td>
</tr>
<tr>
<td>L1 port width</td>
<td>2×16+1×16 B</td>
<td>2×16+1×16 B</td>
<td>2×32+1×32 B</td>
</tr>
<tr>
<td>ADD throughput</td>
<td>1 / cy</td>
<td>1 / cy</td>
<td>1 / cy</td>
</tr>
<tr>
<td>MUL throughput</td>
<td>1 / cy</td>
<td>1 / cy</td>
<td>2 / cy</td>
</tr>
<tr>
<td>FMA throughput</td>
<td>n/a</td>
<td>n/a</td>
<td>2 / cy</td>
</tr>
<tr>
<td>L2-L1 data bus</td>
<td>32 B</td>
<td>32 B</td>
<td>64 B</td>
</tr>
<tr>
<td>L3-L2 data bus</td>
<td>32 B</td>
<td>32 B</td>
<td>32 B</td>
</tr>
<tr>
<td>LLC size</td>
<td>20 MiB</td>
<td>25 MiB</td>
<td>35 MiB</td>
</tr>
<tr>
<td>Main memory</td>
<td>4×DDR3-1600</td>
<td>4×DDR3-1866</td>
<td>4×DDR4-2133</td>
</tr>
<tr>
<td>Peak memory BW</td>
<td>51.2 GB/s</td>
<td>51.2 GB/s</td>
<td>68.3 GB/s</td>
</tr>
<tr>
<td>Load-only BW</td>
<td>43.6 GB/s (85%)</td>
<td>46.1 GB/s (90%)</td>
<td>60.6 GB/s (89%)</td>
</tr>
<tr>
<td>$T_{L3Mcm}$ per CL</td>
<td>3.96 cy</td>
<td>3.05 cy</td>
<td>2.43 cy</td>
</tr>
</tbody>
</table>
Simple streaming benchmark:

```fortran
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A

do j=1,NITER
   do i=1,N
      A(i) = B(i) + C(i) * D(i)
   enddo
   if(.something.that.is.never.true.) then
      call dummy(A,B,C,D)
   endif
enddo
```

- Report performance for different N
- Choose NITER so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all current architectures!

Prevents smarty-pants compilers from doing “clever” stuff
\( A(\cdot) = B(\cdot) + C(\cdot) \times D(\cdot) \) on one Sandy Bridge core (3 GHz)

Are the performance levels plausible?

What about multiple cores?

Do the bandwidths scale?

Pattern! Ineffective instructions

4 W / iteration \( \rightarrow 128 \text{ GB/s} \)

5 W / it. \( \rightarrow 18 \text{ GB/s} \) (incl. write allocate)
$A(:) = B(:) + C(:) \times D(:)$ on one Sandy Bridge core (3 GHz):

Observations and further questions

See later for answers!
The throughput-parallel vector triad benchmark

Every core runs its own, independent triad benchmark

double precision, dimension(,:), allocatable :: A,B,C,D

!$OMP PARALLEL private(i,j,A,B,C,D)
allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
do j=1,NITER
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
enddo
!$OMP END PARALLEL

→ pure hardware probing, no impact from OpenMP overhead
Throughput vector triad on Sandy Bridge socket (3 GHz)

Saturation effect in memory

Scalable BW in L1, L2, L3 cache
Attainable memory bandwidth: Comparing architectures

Intel Broadwell (22 cores) CoD enabled

AMD Naples (24 cores)

Intel Xeon Phi 7210 / KNL

NVIDIA P100 (Pascal)

BW saturation in NUMA domain

Single core does not saturate BW

Pattern! Bandwidth saturation

(c) RRZE 2017
Conclusions from the microbenchmarks

- **Affinity matters!**
  - Almost all performance properties depend on the position of
    - Data
    - Threads/processes
  - Consequences
    - Know where your threads are running
    - Know where your data is

- **Bandwidth bottlenecks are ubiquitous**
“Simple” performance modeling: The Roofline Model

Loop-based performance modeling: Execution vs. data transfer
Example: array summation
Example: dense & sparse matrix-vector multiplication
Example: a 3D Jacobi solver
Model-guided optimization


Prelude: Modeling customer dispatch in a bank

Rooiling door throughput:
\( b_s \) [customers/sec]

Processing capability:
\( P_{\text{peak}} \) [tasks/sec]

Intensity:
\( I \) [tasks/customer]
Prelude: Modeling customer dispatch in a bank

How fast can tasks be processed? $P$ [tasks/sec]

The bottleneck is either

- The service desks (max. tasks/sec): $P_{\text{peak}}$
- The revolving door (max. customers/sec): $I \cdot b_S$

$$P = \min(P_{\text{peak}}, I \cdot b_S)$$

This is the “(naïve) Roofline Model”

- High intensity: $P$ limited by “execution”
- Low intensity: $P$ limited by “bottleneck”
- “Knee” at $P_{\text{peak}} = I \cdot b_S$:
  - Best use of resources

- Roofline is an “optimistic” model: (“light speed”)
The Roofline Model – Basics

Apply this to performance of compute devices

- Maximum processing capability $\rightarrow$ Peak performance: $P_{\text{peak}} \left[ \frac{F}{s} \right]$
- Rate of revolving door $\rightarrow$ Memory bandwidth: $b_S \left[ \frac{B}{s} \right]$
- Workload per customer $\rightarrow$ Computational Intensity: $I \left[ \frac{F}{B} \right]$

Machine model:

$$P_{\text{peak}} = 4 \frac{GF}{s}$$
$$b_S = 10 \frac{GB}{s}$$

Application model: $I$

$$I = \frac{2F}{16B} = 0.125 \frac{F}{B}$$
The Roofline Model – Basics

Compare capabilities of different machines

- RLM always provides upper bound – but is it realistic?
- If code is not able to reach this limit (e.g. contains add operations only) machine parameter need to redefined (e.g. $P_{peak} \rightarrow \frac{P_{peak}}{2}$)

Assuming double precision – for single precision: $P_{peak} \rightarrow 2 \cdot P_{peak}$
The Roofline Model
(a slightly refined version for better in-core prediction)

1. \( P_{\text{max}} = \) Applicable peak performance of a loop, assuming that data comes from the level 1 cache (this is not necessarily \( P_{\text{peak}} \))
   \( \rightarrow \) e.g., \( P_{\text{max}} = 176 \) GFlop/s

2. \( I = \) Computational intensity ("work" per byte transferred) over the slowest data path utilized (code balance \( B_C = I^{-1} \))
   \( \rightarrow \) e.g., \( I = 0.167 \) Flop/Byte \( \rightarrow B_C = 6 \) Byte/Flop

3. \( b_S = \) Applicable peak bandwidth of the slowest data path utilized
   \( \rightarrow \) e.g., \( b_S = 56 \) GByte/s

Expected performance:

\[
P = \min(P_{\text{max}}, I \cdot b_S) = \min\left(P_{\text{max}}, \frac{b_S}{B_C}\right)
\]

[Byte/s] [Byte/Flop]
Every new CPU generation provides incremental improvements.
Example: Estimate $P_{\text{max}}$ of vector triad on Haswell

for (int i=0; i<N; i++) {
    A[i] = B[i] + C[i] * D[i];
}

Minimum number of cycles to process one AVX-vectorized iteration (one core)?

→ Equivalent to 4 scalar iterations

Cycle 1: LOAD + LOAD + STORE
Cycle 2: LOAD + LOAD + FMA + FMA
Cycle 3: LOAD + LOAD + STORE

Answer: 1.5 cycles
Example: Estimate $P_{\text{max}}$ of vector triad on Haswell (2.3 GHz)

for (int i=0; i<N; i++) {
    A[i] = B[i] + C[i] * D[i];
}

What is the performance in GFlops/s per core and the bandwidth in GBytes/s?

One AVX iteration (1.5 cycles) does $4 \times 2 = 8$ flops:

$$\frac{2.3 \times 10^9 \text{ cy/s}}{1.5 \text{ cy}} \cdot \frac{4 \text{ updates}}{2 \text{ flops}} = \frac{12.27 \text{ Gflops}}{s}$$

$$\frac{6.13 \times 10^9 \text{ updates}}{s} \cdot \frac{32 \text{ bytes}}{32 \text{ bytes \ update}} = \frac{196 \text{ Gbyte}}{s}$$
**$P_{\text{max}}$ + bandwidth limitations:** The vector triad

**Vector triad** $A(:) = B(:) + C(:) \cdot D(:)$ on a 2.3 GHz 14-core Haswell chip

Consider full chip (14 cores):

- **Memory bandwidth:** $b_S = 50$ GB/s
- **Code balance** (incl. write allocate): $B_c = (4+1)$ Words / 2 Flops = 20 B/F $\Rightarrow I = 0.05$ F/B

$\Rightarrow I \cdot b_S = 2.5$ GF/s (0.5% of peak performance)

- $P_{\text{peak}} / \text{core} = 36.8$ Gflop/s ($(8+8)$ Flops/cy x 2.3 GHz)
- $P_{\text{max}} / \text{core} = 12.27$ Gflop/s (see prev. slide)

$\Rightarrow P_{\text{max}} = 14 \times 12.27$ Gflop/s = 172 Gflop/s (33% peak)

$$P = \min(P_{\text{max}}, I \cdot b_S) = \min(172, 2.5) \text{ GFlop/s} = 2.5 \text{ GFlop/s}$$

(c) RRZE 2017

Node-Level Performance Engineering
A not so simple Roofline example

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

in single precision on a 2.2 GHz Sandy Bridge socket @ “large” N

\[ P = \min(P_{\text{max}}, I \cdot b_S) \]

\( P \) (better loop code)

Machine peak (ADD+MULT)
Out of reach for this code

\( P \) (worst loop code)

ADD peak (best possible code)

3-cycle latency per ADD if not pipelined

How do we get these? \( \Rightarrow \) See next!

\( I = 1 \text{ flop } / 4 \text{ byte } \) (SP!)
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:

→ 1/24 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:

→ 1/8 of ADD peak
Applicable peak for the summation loop

SIMD-vectorized, 3-way unrolled

LOAD \[r1.0,\ldots, r1.7]\ \leftarrow \ [0,\ldots, 0] \\
LOAD \[r2.0,\ldots, r2.7]\ \leftarrow \ [0,\ldots, 0] \\
LOAD \[r3.0,\ldots, r3.7]\ \leftarrow \ [0,\ldots, 0] \\
i \leftarrow 1

\textbf{loop:}

LOAD \[r4.0,\ldots, r4.7]\ \leftarrow \ [a(i),\ldots, a(i+7)] \\
LOAD \[r5.0,\ldots, r5.7]\ \leftarrow \ [a(i+8),\ldots, a(i+15)] \\
LOAD \[r6.0,\ldots, r6.7]\ \leftarrow \ [a(i+16),\ldots, a(i+23)] \\

\text{ADD } \ r1 \leftarrow r1 + r4 \\
\text{ADD } \ r2 \leftarrow r2 + r5 \\
\text{ADD } \ r3 \leftarrow r3 + r6 \\

i+=24 \ \rightarrow \ ? \ \text{loop} \\
\text{result } \leftarrow r1.0 + r1.1 + \ldots + r3.6 + r3.7 \\

\text{ADD pipes utilization:} \\
\rightarrow \text{ADD peak}
Input to the roofline model

... on the example of

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

in single precision

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

pipeline depth: 3 cy (ADD)

8-way SIMD, 8 cores

maximum memory bandwidth 40 GB/s

worst code: \( P = 5.9 \text{ GF/s} \) (core bound)

better code: \( P = 10 \text{ GF/s} \) (memory bound)

10 GF/s

measurement

architecture

code analysis:
1 ADD + 1 LOAD

analysis
Prerequisites 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!
  - Either the limit is core execution or it is data transfer

- Slowest limiting factor “wins”; all others are assumed to have no impact

- Latency effects are ignored: perfect data streaming, “steady-state” execution, no start-up effects
Case study: 
Dense Matrix Vector Multiplication

Where does the data come from?
Example: Dense matrix-vector multiplication in DP (AVX)

\[
\text{do } c = 1 , C \\
\quad \text{do } r = 1 , R \\
\quad \quad y(r) = y(r) + A(r,c) \times x(c) \\
\quad \text{enddo} \\
\text{enddo}
\]

- Assume \( C = R \approx 10,000 \)
- Applicable peak performance?
- Relevant data path?
- Computational Intensity?

\[
\text{do } c = 1 , C \\
\quad \text{tmp = } x(c) \\
\quad \text{do } r = 1 , R \\
\quad \quad y(r) = y(r) + A(r,c) \times \text{tmp} \\
\quad \text{enddo} \\
\text{enddo}
\]
Performance drops as number of rows (inner loop length) increases.

Does computational intensity change?!

Intel Xeon E5 2695 v3 (Haswell-EP), 2.3 GHz, CoD mode, Core $P_{\text{peak}}=36.8$ GF/s, Caches: 32 KB / 256 KB / 35 MB, PageSize: 2 MB; ifort V15.0.1.133; $b_S = 32$ Gbyte/s
DMVM data traffic analysis

do c = 1 , C
    tmp=x(c)
do r = 1 , R
    y(r)=y(r) + A(r,c)* tmp
  enddo
enddo

tmp stays in a register during inner loop
A(:, :) is loaded from memory – no data reuse

y(:) is loaded and stored in each outer iteration → for c>1 update y(:) in cache

y(:) may not fit in innermost cache → more traffic from lower level caches for larger R

Roofline analysis: Distinguish code balance in memory ($B_{mem}^C$) from code balance in relevant cache level(s) ($B_{L3}^C$, $B_{L2}^C$, ...)!
Exceeding inner cache size: 
$\Rightarrow (8+8)$ Byte for LD + ST on $y$

$size(y(1:R)) = 16$ KB

$size(y(1:R)) = 160$ KB

Data traffic [B/Iteration] vs. $R$

$B_{C}^{L2} = 24B/2F$

$B_{C}^{L3} = 24B/2F$

$B_{C}^{mem} = 8B/2F$

$B_{C}^{L3} = 8B/2F$

$C=10^4$

$c$ RRZE 2017 Node-Level Performance Engineering
Reducing traffic by blocking

\[ A(r,c) \]

\[
\begin{align*}
do & \ c = 1 \ , \ C \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = x(c) \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = y(r) + A(r,c) \times \text{tmp} \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = \text{y}(r) \\
\end{align*}
\]

\[
\begin{align*}
do & \ b = 1 \ , \ R \ , \ R_b \\
& \ rbS = rb \\
& \ rbE = \min((rb+R_b-1), R) \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = y(r) \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = A(r,c) \times x(c) \\
& \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = \text{y}(rbS:rbE) \\
\end{align*}
\]
Reducing traffic by blocking

- **LHS only updated once** in some cache level if blocking is applied
- **Price**: RHS is loaded multiple times instead of once!
  - How often? $\rightarrow R / R_b$ times
  - RHS traffic: $C \times R / R_b$
  - LHS traffic: $2 \times R$
  - Matrix: $R \times R$

- Overall: $R \times \left(\frac{C}{R_b} + 2 + R\right) \approx R^2$ if $R, R_b \gg 1$ and $C = R$
- Without blocking: $R \times \left(\frac{C}{R} + 2C + R\right) \approx 3R^2$ if $R, R_b \gg 1$ and $C = R$
DMVM (DP) – Reducing traffic by inner loop blocking

- “1D blocking” for inner loop
- Blocking factor $R_b \leftrightarrow$ cache level

```plaintext
do rb = 1 , R , $R_b$
    rbS = rb
    rbE = min((rb+$R_b$-1), R)
    do c = 1 , C
        do r = rbS , rbE
            $y(r)$ = $y(r)$ + $A(r,c)$*$x(c)$
        enddo
    enddo
enddo
```

→ Fully reuse subset of $y(rbS:rbE)$ from L1/L2 cache
DMVM (DP) – Validation of blocking optimization

Node-Level Performance Engineering

(c) RRZE 2017
DMVM (DP) – OpenMP parallelization

```c
!$omp parallel do reduction(+:y)
  do c = 1 , C
    do r = 1 , R
      y(r) = y(r) + A(r,c) * x(c)
    enddo
  enddo
!$omp end parallel do
```

Plain code:

```c
!$omp parallel do private(rbS,rbE)
  do rb = 1 , R , R_b
    rbS = rb
    rbE = min((rb+R_b-1), R)
  enddo
  do c = 1 , C
    do r = rbS , rbE
      y(r) = y(r) + A(r,c) * x(c)
    enddo
  enddo
!$omp end parallel do
```

Blocked code:
DMVM (DP) – OpenMP parallelization & saturation
“Using more cores can heal bad single-core performance”

Roofline limit
\[ B_C = 4 \text{ Byte/Flop} \]
\[ b_S = 32 \text{GB/s} \]

Pattern! Bandwidth saturation

memory traffic unchanged
\rightarrow saturation unchanged!

saturation influenced by serial performance

So, is blocking useless?
\rightarrow \text{NO (see later)}

Can we do anything to improve \( B_C^{\text{mem}} \)?
\rightarrow \text{NO, not here}

Intel Xeon E5 2695 v3 (Haswell-EP)
2.3 GHz base clock speed, \( b_S = 32 \text{ GB/s} \)

blocking good for single thread performance (reduced in-cache traffic)
Conclusions from the dMVM example

- We have found the reasons for the breakdown of single-core performance with growing number of matrix rows
  - LHS vector fitting in different levels of the cache hierarchy
  - Validated theory by performance counter measurements

- **Inner loop blocking was employed to improve code balance in L3 and/or L2**
  - Validated by performance counter measurements

- **Blocking led to better single-threaded performance**

- **Saturated performance unchanged (as predicted by Roofline)**
  - Because the problem is still small enough to fit the LHS at least into the L3 cache
Typical code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code (e.g., Perl → Fortran)
2. Increase intensity to make better use of BW bottleneck (e.g., loop blocking → see later)
3. Increase intensity and go from memory-bound to core-bound (e.g., temporal blocking)
4. Hit the core bottleneck by good serial code (e.g., -fno-alias → see later)
5. Shift \( P_{\text{max}} \) by accessing additional hardware features or using a different algorithm/implementation (e.g., scalar → SIMD)
Using Roofline for monitoring “live” jobs on a cluster
Based on measured BW and Flop/s data via likwid-perfctr

Where are the “good” and the “bad” jobs in this diagram?
Case study: A Jacobi smoother

The basic performance properties in 2D
Layer conditions
Optimization by spatial blocking
Stencil schemes

- Stencil schemes frequently occur in PDE solvers on regular lattice structures
- Basically it is a sparse matrix vector multiply (spMVM) embedded in an iterative scheme (outer loop)
- but the regular access structure allows for matrix free coding

```plaintext
do iter = 1, max_iterations

  Perform sweep over regular grid: y( :) ← x ( : )

  Swap y ↔ x

endo
```

- Complexity of implementation and performance depends on
  - update scheme, e.g. Jacobi-type, Gauss-Seidel-type, …
  - spatial extent, e.g. 7-pt or 25-pt in 3D,…
Jacobi-type 5-pt stencil in 2D

\begin{verbatim}
  do k=1,kmax
    do j=1,jmax
      y(j,k) = const * ( x(j-1,k) + x(j+1,k) &
                        + x(j,k-1) + x(j,k+1) )
    enddo
  enddo
\end{verbatim}

Appropriate performance metric: “Lattice Updates per second” [LUP/s]

(here: Multiply by 4 FLOP/LUP to get FLOP/s rate)
Jacobi 5-pt stencil in 2D: data transfer analysis

```
 do k=1,kmax
   do j=1,jmax
     y(j,k) = const * ( x(j-1,k) + x(j+1,k) &
                        + x(j,k-1) + x(j,k+1) )
   enddo
 enddo
```

Naive balance (incl. write allocate):

- \( x(:, :) \) : 3 LD +
- \( y(:, :) \) : 1 ST + 1 LD

\( B_C = 5 \text{ Words} / \text{LUP} = 40 \text{ B} / \text{LUP} \) (assuming double precision)
Questions:

1. How to achieve 24 B/LUP also for large $j_{\text{max}}$?

2. How to sustain >600 MLUP/s for $j_{\text{max}} > 10^4$?
Case study: A Jacobi smoother

The basics in two dimensions
Layer conditions
Optimization by spatial blocking
Analyzing the data flow

Worst case: Cache not large enough to hold 3 layers (rows) of grid (assume “Least Recently Used” replacement strategy)

```
x(0:jmax+1,0:kmax+1)
```

- Hit: 1 cache miss
- Miss: 2 cache misses
- Halo cells
Analyzing the data flow

Worst case: Cache not large enough to hold 3 layers (rows) of grid (+assume “Least Recently Used“ replacement strategy)
Analyzing the data flow

Reduce inner \((j-)\) loop dimension successively

Best case: 3 “layers” of grid fit into the cache!
Analyzing the data flow: Layer condition

2D 5-pt Jacobi-type stencil

\[
\begin{align*}
\text{do } & \text{k=1,kmax} \\
& \text{do } \text{j=1,jmax} \\
& \quad y(j,k) = \text{const} \times (x(j-1,k) + x(j+1,k) \& \\
& \quad \quad + x(j,k-1) + x(j,k+1)) \\
\end{align*}
\]

\[3 \times jmax \times 8B < \text{CacheSize}/2\]

“Layer condition”

Layer condition:
• Does not depend on outer loop length (kmax)
• No strict guideline (cache associativity – data traffic for y not included)
• Needs to be adapted for other stencils (e.g., 3D 7-pt stencil)
Analyzing the data flow: Layer condition (2D 5-pt Jacobi)

\[
\text{do } k=1, k_{\text{max}} \\
\quad \text{do } j=1, j_{\text{max}} \\
\quad \quad y(j,k) = \text{const} \times (x(j-1,k) + x(j+1,k) \& + x(j,k-1) + x(j,k+1)) \\
\quad \text{enddo} \\
\text{enddo}
\]

\[ B_C = 24 \text{ B} / \text{LUP} \]

\[ 3 \times j_{\text{max}} \times 8B < \text{CacheSize} / 2 \]

Layer condition fulfilled?

\[
\text{do } k=1, k_{\text{max}} \\
\quad \text{do } j=1, j_{\text{max}} \\
\quad \quad y(j,k) = \text{const} \times (x(j-1,k) + x(j+1,k) \& + x(j,k-1) + x(j,k+1)) \\
\quad \text{enddo} \\
\text{enddo}
\]

\[ B_C = 40 \text{ B} / \text{LUP} \]

\[ y: (1 \text{ LD} + 1 \text{ ST}) / \text{LUP} \]

\[ x: 1 \text{ LD} / \text{LUP} \]

\[ x: 3 \text{ LD} / \text{LUP} \]
Analyzing the data flow: Layer condition (2D 5-pt Jacobi)

- Establish layer condition for all domain sizes
- Idea: Spatial blocking
  - Reuse elements of \( x() \) as long as they stay in cache
  - Sweep can be executed in any order, e.g. compute blocks in j-direction

→ “Spatial Blocking” of j-loop:

```fortran
    do jb=1,jmax,jblock ! Assume jmax is multiple of jblock
        do k=1,kmax
            do j= jb, (jb+jblock-1) ! Length of inner loop: jblock
                y(j,k) = const * (x(j-1,k) + x(j+1,k) &
                                + x(j,k-1) + x(j,k+1) )
            enddo
        enddo
    enddo
```

New layer condition (blocking)

\[ 3 \times j\text{block} \times 8B < \frac{\text{CacheSize}}{2} \]

→ Determine for given CacheSize an appropriate j\text{block} value:

\[ j\text{block} < \frac{\text{CacheSize}}{48 \text{ B}} \]
Establish the layer condition by blocking

Split up domain into subblocks:

e.g. block size = 5
Establish the layer condition by blocking

Additional data transfers (overhead) at block boundaries!
Establish layer condition by spatial blocking

\[ j_{\text{block}} < \text{CacheSize} / 48 \text{ B} \]

Which cache to block for?

![Graph showing performance vs. \( j_{\text{max}} \) and \( k_{\text{max}} \)]

- L1: 32 KB
- L2: 256 KB
- L3: 25 MB

Intel Xeon E5-2690 v2 (“IvyBridge”@3 GHz)

Intel Compiler ifort V13.1

\[ j_{\text{max}} = k_{\text{max}} \]
\[ j_{\text{max}} \times k_{\text{max}} = \text{const} \]
Layer condition & spatial blocking: Memory code balance

Main memory access is not reason for different performance (but L3 access is!)

Blocking factor (CS=25 MB) still a little too large

Measured main memory code balance ($B_C$)

Intel Compiler
ifort V13.1
Intel Xeon E5-2690 v2 ("IvyBridge"@3 GHz)
Jacobi Stencil – OpenMP parallelization

```c
!$OMP PARALLEL DO SCHEDULE(STATIC)
do k=1,kmax
  do j=1,jmax
    y(j,k) = 1/4.*(x(j-1,k) + x(j+1,k) &
                 + x(j,k-1) + x(j,k+1))
  enddo
enddo
```

**Basic guideline:**
Parallelize outermost loop

**Equally large chunks in k-direction** → “Layer condition” for each thread

**“Layer condition”**:  
\[ 3 \times \text{imax} \times 8B < \frac{CS_t}{2} \]

\( CS_t = \text{cache per thread} \)
OpenMP parallelization and blocking for shared cache

Example: 2D 5-point stencil on Sandy Bridge 8-core, 20 MB L3

- Optimal jblock for 1 thread is too small for multiple threads
- Smaller but constant jblock works for few threads but not for all
- Optimal blocking for shared cache requires adaptive block size

\[ j_{\text{max}} = 1.2 \times 10^6 \]

Pattern! Excess data volume
Stencil shapes and layer conditions

(a) **Long-range** \( r = 2 \): 5 layers \((2r + 1)\)

(b) **Long-range** \( r = 2 \) with gaps: 6 layers \((2 \text{ per populated row})\)

(c) **Asymmetric**: 3 layers

(d) **2D box**: 3 layers
Conclusions from the Jacobi example

- **We have made sense of the memory-bound performance vs. problem size**
  - “Layer conditions” lead to predictions of code balance
  - “What part of the data comes from where” is a crucial question
  - The model works only if the bandwidth is “saturated”
    - In-cache modeling is more involved

- **Avoiding slow data paths == re-establishing the most favorable layer condition**

- **Improved code showed the speedup predicted by the model**

- **Optimal blocking factor can be estimated**
  - Be guided by the cache size the layer condition
  - No need for exhaustive scan of “optimization space”

- **Food for thought**
  - Multi-dimensional loop blocking – would it make sense?
  - Can we choose a “better” OpenMP loop schedule?
  - What would change if we parallelized inner loops?
Case study: Sparse Matrix Vector Multiplication

An example for what to do if the Roofline model isn’t quite so predictive
Sparse Matrix Vector Multiplication (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”: $N_{nz} \sim N_r$
SpMVM characteristics

- For large problems, spMVM is inevitably memory-bound
  - Intra-socket saturation effect on modern multicores

- SpMVM is easily parallelizable in shared and distributed memory

- Data storage format is crucial for performance properties
  - Most useful general format on CPUs: Compressed Row Storage (CRS)
  - Depending on compute architecture
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: $N_r$)
Case study: Sparse matrix-vector multiply

- **Strongly memory-bound for large data sets**
  - Streaming, with partially indirect access:

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

- Usually many spMVMs required to solve a problem

- **Now let’s look at some performance measurements…**
Performance characteristics

- Strongly memory-bound for large data sets → saturating performance across cores on the chip
- Performance seems to depend on the matrix

- Can we explain this?
- Is there a “light speed” for spMVM?
- Optimization?
Example: SpMVM node performance model

- Sparse MVM in double precision w/ CRS data storage:
  
  ```
  do i = 1,N_r
    do j = row_ptr(i), row_ptr(i+1) - 1
      C(i) = C(i) + val(j) * B(col_idx(j))
    enddo
  enddo
  ```

- DP CRS comp. intensity
  - $\alpha$ quantifies traffic for loading RHS
    - $\alpha = 0 \Rightarrow$ RHS is in cache
    - $\alpha = 1/N_{nzr} \Rightarrow$ RHS loaded once
    - $\alpha = 1 \Rightarrow$ no cache
    - $\alpha > 1 \Rightarrow$ Houston, we have a problem!
  - “Expected” performance = $b_s \times I_{CRS}$
  - Determine $\alpha$ by measuring performance and actual memory traffic
    - Maximum memory BW may not be achieved with spMVM

\[
I_{CRS}^{DP} = \frac{2}{8 + 4 + 8\alpha + 16/N_{nzr}} \text{ flops/byte}
\]

“best” in-memory intensity:

$I = \frac{1}{6} F/B$, or $B_C^{mem} = 6 \text{ B/F}$
Determine RHS traffic

\[ I_{CRS}^{DP} = \frac{2}{8 + 4 + 8\alpha + 16/N_{nzr}} \text{ flops} = \frac{N_{nz} \cdot 2 \text{ flops}}{V_{meas}} \]

- \( V_{meas} \) is the measured overall memory data traffic (using, e.g., likwid-perfctr)
- Solve for \( \alpha \):
  \[
  \alpha = \frac{1}{4} \left( \frac{V_{meas}}{N_{nz} \cdot 2 \text{ bytes}} - 6 - \frac{8}{N_{nzr}} \right)
  \]

Example: kkt_power matrix from the UoF collection on one Intel SNB socket
- \( N_{nz} = 14.6 \cdot 10^6, N_{nzr} = 7.1 \)
- \( V_{meas} \approx 258 \text{ MB} \)
- \( \Rightarrow \alpha = 0.43, \alpha N_{nzr} = 3.1 \)
- \( \Rightarrow \) RHS is loaded 3.1 times from memory
- and:
  \[
  \frac{I_{CRS}^{DP}(1/N_{nzr})}{I_{CRS}^{DP}(\alpha)} = 1.15
  \]

15% extra traffic \( \Rightarrow \) optimization potential!
Now back to the start…

- $b_S = 39 \text{ GB/s}$
- $B_c^{\text{min}} = 6 \text{ B/F}$
- Maximum spMVM performance:
  - $P_{\text{max}} = 6.5 \text{ GF/s}$
  - $\rightarrow$ DLR1 causes minimum code balance!

- sAMG matrix code balance:
  - $B_c \leq \frac{b_S}{4.5 \text{ GF/s}} = 8.7 \text{ B/F}$

- Why is this only an upper limit?
- What is the next step?
- Could we have predicted this qualitative difference?
Sparse matrix testcases

“DLR1” (A. Basermann, DLR)
Adjoint problem computation (turbulent transonic flow over a wing) with the TAU CFD system of the German Aerospace Center (DLR)
Avg. non-zeros/row ~150

“sAMG” (K. Stüben, FhG-SCAI)
Matrix from FhG’s adaptive multigrid code sAMG for the irregular discretization of a Poisson problem on a car geometry.
Avg. non-zeros/row ~ 7
Conclusion from Roofline analysis

- The roofline model does not “work” for spMVM due to the RHS traffic uncertainties
- We have “turned the model around” and measured the actual memory traffic to determine the RHS overhead
- Result indicates:
  1. how much actual traffic the RHS generates
  2. how efficient the RHS access is (compare BW with max. BW)
  3. how much optimization potential we have with matrix reordering

Consequence: Modeling is not always 100% predictive. It’s all about learning more about performance properties!
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
  - It is not sufficient to measure single-core STREAM to make it work
  - Only increased “pressure” on the memory interface can saturate the bus → need more cores!

- **In-cache performance is not correctly predicted**

- **The ECM performance model gives more insight:**
  
Coding for SINGLE INSTRUCTION MULTIPLE DATA processing

Basics
Data types and data layout
Compiler support (Intel)
SIMD is an in-core feature
SIMD processing – Basics

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

- x86 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands
  - AVX512: you get it.

- Adding two registers holding double precision floating point operands

Scalar execution: 
R2 ← ADD [R0,R1]

SIMD execution: 
V64ADD [R0,R1] → R2
Data types in 32-byte SIMD registers

- Supported data types depend on actual SIMD instruction set

![Diagram showing supported data types in 32-byte SIMD registers](image)
SIMD style structure of arrays processing

A(:) xmm0
B(:) xmm1
C(:) xmm2
D(:) xmm3

real, dimension(:) :: A,B,C,D,R

R(:)
Data layout can have significant impact on the efficiency of SIMD processing!
Vectorization compiler options (Intel)

- The compiler will vectorize starting with `-O2`.
- To enable specific SIMD extensions use the `-x` option:
  - `-xSSE2` vectorize for SSE2 capable machines

Available SIMD extensions (non-exhaustive):
SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, CORE-AVX2, CORE-AVX512, MIC-AVX512

- `-xCORE-AVX2` on Haswell/Broadwell
- `-xMIC-AVX512` on Knights Landing

Recommended option:
- `-xHost` will optimize for the architecture you are compiling on
Vectorization compiler options

- **Controlling non-temporal stores (part of the SIMD extensions)**

  - `-opt-streaming-stores` **always** | **auto** | **never**

    - **always**  use NT stores, assume application is memory bound (use with caution!)
    - **auto**   compiler decides when to use NT stores
    - **never** do not use NT stores unless activated by source code directive
Vectorization source code directives

- Fine-grained control of loop vectorization
  - Use !DEC$ (Fortran) or #pragma (C/C++) sentinel to start a compiler directive

- #pragma vector always
  vectorize even if it seems inefficient (hint!)

- #pragma novector
  do not vectorize even if possible

- #pragma vector nontemporal
  use NT stores when allowed (i.e. alignment conditions are met)

- #pragma vector aligned
  specifies that all array accesses are aligned to 16-byte boundaries (DANGEROUS! You must not lie about this!)
User mandated vectorization

- Since Intel Compiler 12.0 the `simd` pragma is available
- `#pragma simd` enforces vectorization where the other pragmas fail
- Prerequisites:
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs
- There are additional clauses: `reduction`, `vectorlength`, `private`
- Refer to the compiler manual for further details

```cpp
#pragma simd reduction(+:x)
for (int i=0; i<n; i++) {
    x = x + A[i];
}
```

- **NOTE:** Using the `#pragma simd` the compiler may generate incorrect code if the loop violates the vectorization rules!
x86 Architecture: 
SIMD and Alignment

- **Alignment issues**
  - Alignment of arrays with SSE (AVX) should be on 16-byte (32-byte) boundaries to allow packed aligned loads and NT stores *(for Intel processors)*
    - **AMD has a scalar nontemporal store instruction**
  - Otherwise the compiler will revert to unaligned loads and not use NT stores – even if you say `vector nontemporal`
  - Modern x86 CPUs have less (not zero) impact for misaligned LD/ST, but *Xeon Phi (KNC) relies heavily on it!*
  - How is manual alignment accomplished?

- **Dynamic allocation of aligned memory** *(align = alignment boundary)*:

```c
#define _XOPEN_SOURCE 600
#include <stdlib.h>

int posix_memalign(void **ptr, 
                   size_t align, 
                   size_t size);
```
SIMD is an in-core feature!

DP sum reduction (single core)

\[
\begin{align*}
\text{do } & \ i = 1, N \\
& \quad s = s + A(i) \\
\text{enddo}
\end{align*}
\]

Intel Broadwell EP 2.3 GHz

Intel KNL 1.3 GHz
Reading x86 assembly code and exploiting SIMD parallelism

Understanding SIMD execution by inspecting assembly code
SIMD vectorization how-to
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):
  \[ \text{icc -S -masm=intel -O3 -xHost triad.c -o a.out} \]

- Disassemble Executable:
  \[ \text{objdump -d ./a.out | less} \]

The x86 ISA is documented in:
- Intel Software Development Manual (SDM) 2A and 2B
Basics of the x86-64 ISA

- Instructions have 0 to 4 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**:  mul, add, mov
**Modifier**:  nontemporal (nt), unaligned (u), aligned (a), high (h)
**Width**:  scalar (s), packed (p)
**Data type**:  single (s), double (d)
for (int i = 0; i < length; i++) {
    A[i] = B[i] + D[i] * C[i];
}

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

Assembly code (-O1):

```
for (int i = 0; i < length; i++) {
    A[i] = B[i] + D[i] * C[i];
}

CLANG
LBB0_3:
movsd    xmm0, [rdx]
mulsd    xmm0, [rcx]
addsd    xmm0, [rsi]
movsd    [rax], xmm0
add       rsi, 8
add       rdx, 8
add       rcx, 8
add       rax, 8
dec       edi
jne       LBB0_3

 ICC
 .L4:
movsd    xmm0, [r12+rax*8]
mulsd    xmm0, [r13+rax*8]
addsd    xmm0, [r14+rax*8]
movsd    [r15+rax*8], xmm0
inc       rax
cmp       rax, rbx
jl        ..B1.6

 GCC
 .L4:
movsd    xmm0, [rbx+rbx]
mulsd    xmm0, [r12+rax]
addsd    xmm0, [r13+rbx]
movsd    [rbp+0+rax], xmm0
add       rax, 8
jmp       ..B1.6
jne       .L4
```

7 instructions per loop iteration
Case Study: Vector Triad (DP) -O3 -xHost
+ #pragma vector aligned

..B1.7:

```
vmovupd  ymm0, [r15+rcx*8]
vmovupd  ymm4, [32+r15+rcx*8]
vmovupd  ymm7, [64+r15+rcx*8]
vmovupd  ymm10, [96+r15+rcx*8]
vmulpd  ymm2, ymm0, [rdx+rcx*8]
vmulpd  ymm5, ymm4, [32+rdx+rcx*8]
vmulpd  ymm8, ymm7, [64+rdx+rcx*8]
vmulpd  ymm11, ymm10, [96+rdx+rcx*8]
vaddpd  ymm3, ymm2, [r14+rcx*8]
vaddpd  ymm6, ymm5, [32+r14+rcx*8]
vaddpd  ymm9, ymm8, [64+r14+rcx*8]
vaddpd  ymm12, ymm11, [96+r14+rcx*8]
vmovupd  [r13+rcx*8], ymm3
vmovupd  [32+r13+rcx*8], ymm6
vmovupd  [64+r13+rcx*8], ymm9
vmovupd  [96+r13+rcx*8], ymm12
add       rcx, 16
cmp       rcx, rsi
jb         ..B1.7
```

1.19 instructions per loop iteration
Case Study: Vector Triad (DP) -O3 -xHost 
#pragma vector aligned on Haswell-EP

..B1.7:

```
vmovupd  ymm2, [r15+rcx*8]
vmovupd  ymm4, [32+r15+rcx*8]
vmovupd  ymm6, [64+r15+rcx*8]
vmovupd  ymm8, [96+r15+rcx*8]
vmovupd  ymm0, [rdx+rcx*8]
vmovupd  ymm3, [32+rdx+rcx*8]
vmovupd  ymm5, [64+rdx+rcx*8]
vmovupd  ymm7, [96+rdx+rcx*8]
vfmadd213pd ymm2, ymm0, [r14+rcx*8]
vfmadd213pd ymm4, ymm3, [32+r14+rcx*8]
vfmadd213pd ymm6, ymm5, [64+r14+rcx*8]
vfmadd213pd ymm8, ymm7, [96+r14+rcx*8]
vmovupd  [r13+rcx*8], ymm2
vmovupd  [32+r13+rcx*8], ymm4
vmovupd  [64+r13+rcx*8], ymm6
vmovupd  [96+r13+rcx*8], ymm8
add rcx, 16
cmp rcx, rsi
jb ..B1.7
```

On X86 ISA instruction are converted to so-called µops (elementary ops like load, add, mult). For performance the number of µops is important.

```
23 uops vs. 27 µops (AVX)
```

1.19 instructions per loop iteration
How to leverage SIMD: your options

Alternatives:
- The compiler does it for you (but: aliasing, alignment, language)
- Compiler directives (pragmas)
- Alternative programming models for compute kernels (OpenCL, ispc)
- Intrinsics (restricted to C/C++)
- Implement directly in assembler

To use intrinsics the following headers are available:
- `xmmintrin.h` (SSE)
- `pmmintrin.h` (SSE2)
- `immintrin.h` (AVX)
- `x86intrin.h` (all extensions)

```c
for (int j=0; j<size; j+=16) {
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}
```
Rules for vectorizable loops

1. Inner loop
2. Countable (loop length can be determined at loop entry)
3. Single entry and single exit
4. Straight line code (no conditionals)
5. No (unresolvable) read-after-write data dependencies
6. No function calls (exception intrinsic math functions)

Better performance with:
1. Simple inner loops with unit stride (contiguous data access)
2. Minimize indirect addressing
3. Align data structures to SIMD width boundary
4. In C use the restrict keyword and/or const qualifiers and/or compiler options to rule out array/pointer aliasing
Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes
First touch placement policy
ccNUMA performance problems
“The other affinity” to care about

- ccNUMA:
  - Whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)

- How do we make sure that memory access is always as "local" and "distributed" as possible?

- Page placement is implemented in units of OS pages (often 4kB, possibly more)
Intel Broadwell EP node
2 chips, 2 sockets, 11 cores per ccNUMA domain (CoD mode)

- **ccNUMA map**: Bandwidth penalties for remote access
  - Run 11 threads per ccNUMA domain (half chip)
  - Place memory in different domain → 4x4 combinations
  - STREAM copy benchmark using standard stores

![STREAM triad performance graph]

(c) RRZE 2017
numactl as a simple ccNUMA locality tool:
How do we enforce some locality of access?

- **numactl** can influence the way a binary maps its memory pages:

  ```
  numactl --membind=<nodes> a.out  # map pages only on <nodes>
  --preferred=<node> a.out  # map pages on <node>
  # and others if <node> is full
  --interleave=<nodes> a.out  # map pages round robin across
  # all <nodes>
  ```

- **Examples:**

  ```
  for m in `seq 0 3`; do
    for c in `seq 0 3`; do
      env OMP_NUM_THREADS=8 \ 
      numactl --membind=$m --cpunodebind=$c ./stream
    enddo
  enddo

  env OMP_NUM_THREADS=4 numactl --interleave=0-3 \ 
  likwid-pin -c N:0,4,8,12 ./stream
  ```

- **But what is the default without numactl?**
ccNUMA default memory locality

- "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
  - This might be a problem, see later

- 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
Coding for ccNUMA data locality

- Most simple case: explicit initialization

```
integer,parameter :: N=10000000
double precision A(N), B(N)

A=0.d0

!$OMP parallel do
do i = 1, N
   B(i) = function ( A(i) )
end do
!$OMP end parallel do
```

```
integer,parameter :: N=10000000
double precision A(N), B(N)

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
   A(i)=0.d0
end do
!$OMP end do
!$OMP end parallel
```

- Most simple case: explicit initialization
Coding for ccNUMA data locality

- Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O

```fortran
integer, parameter :: N=10000000
double precision A(N), B(N)

READ(1000) A

 !$OMP parallel
 !$OMP do schedule(static)
 do i = 1, N
   A(i)=0.d0
 end do
 !$OMP end do
 !$OMP single
 READ(1000) A
 !$OMP end single
 !$OMP parallel
 !$OMP do schedule(static)
 do i = 1, N
   B(i) = function ( A(i) )
 end do
 !$OMP end do
 !$OMP end parallel
```

- Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O

```fortran
integer, parameter :: N=10000000
double precision A(N), B(N)

 !$OMP parallel
 !$OMP do schedule(static)
 do i = 1, N
   A(i)=0.d0
 end do
 !$OMP end do
 !$OMP single
 READ(1000) A
 !$OMP end single
 !$OMP parallel
 !$OMP do schedule(static)
 do i = 1, N
   B(i) = function ( A(i) )
 end do
 !$OMP end do
 !$OMP end parallel
```
Coding for Data Locality

- **Required condition:** OpenMP loop schedule of initialization must be the same as in all computational loops
  - Only choice: `static`! Specify explicitly on all NUMA-sensitive loops, just to be sure…
  - Imposes some constraints on possible optimizations (e.g. load balancing)
  - If dynamic scheduling/tasking is unavoidable, more advanced methods may be in order

- **C++:** Arrays of objects and `std::vector<>` are by default initialized sequentially
  - STL allocators provide an elegant solution

- **Newer Linux kernels:** Automatic page migration may fix the issue automatically
  - Works on a timescale of 1-10 seconds
  - Observable symptom: Code gets faster the longer you run it
Coding for Data Locality: 
*Placement of static arrays or arrays of objects*

- Don't forget that constructors tend to touch the data members of an object. Example:

```cpp
class D {
    double d;
public:
    D(double _d=0.0) throw() : d(_d) {}
    inline D operator+(const D& o) throw() {
        return D(d+o.d);
    }
    inline D operator*(const D& o) throw() {
        return D(d*o.d);
    }
    ...
};

→ placement problem with
D* array = new D[1000000];
```
Coding for Data Locality:  
Parallel first touch for arrays of objects

- **Solution: Provide overloaded** `D::operator new[]`

  ```cpp
  void* D::operator new[](size_t n) {
      char *p = new char[n];    // allocate
  
      size_t i,j;
      #pragma omp parallel for private(j) schedule(...) 
      for(i=0; i<n; i += sizeof(D))
          for(j=0; j<sizeof(D); ++j)
              p[i+j] = 0;
      return p;
  }

  void D::operator delete[](void* p) throw() {
      delete [] static_cast<char*>(p);
  }
  ```

- **Placement of objects is then done automatically by the C++ runtime via “placement new”**
template <class T> class NUMA_Allocator {
public:
    T* allocate(size_type numObjects, const void *localityHint=0) {
        size_type ofs,len = numObjects * sizeof(T);
        void *m = malloc(len);
        char *p = static_cast<char*>(m);
        int i,pages = len >> PAGE_BITS;
        #pragma omp parallel for schedule(static) private(ofs)
        for(i=0; i<pages; ++i) {
            ofs = static_cast<size_t>(i) << PAGE_BITS;
            p[ofs]=0;
        }
        return static_cast<pointer>(m);
    }
}

Application:
vector<double,NUMA_Allocator<double> > x(10000000)
Diagnosing Bad Locality

- If your code is cache bound, you might not notice any locality problems

- Otherwise, bad locality limits scalability at very low CPU numbers (whenever a node boundary is crossed)
  - If the code makes good use of the memory interface

- Running with `numactl --interleave` might give you a hint
  - See later

- Consider using performance counters
  - LIKWID-perfctr can be used to measure nonlocal memory accesses
  - Example for Intel Westmere dual-socket system (Core i7, hex-core):

    ```bash
    env OMP_NUM_THREADS=12 likwid-perfctr -g MEM -C N:0-11 ./a.out
    ```
Using performance counters for diagnosing bad ccNUMA access locality

- Intel Westmere EP node (2x6 cores):

<table>
<thead>
<tr>
<th>Metric</th>
<th>core 0</th>
<th>core 1</th>
<th>core 6</th>
<th>core 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [s]</td>
<td>0.730168</td>
<td>0.733754</td>
<td>0.732808</td>
<td>0.732943</td>
</tr>
<tr>
<td>CPI</td>
<td>10.4164</td>
<td>10.2654</td>
<td>10.5002</td>
<td>10.7641</td>
</tr>
<tr>
<td>Memory bandwidth [MBytes/s]</td>
<td>11880.9</td>
<td>0</td>
<td>...</td>
<td>11732.4</td>
</tr>
<tr>
<td>Remote Read BW [MBytes/s]</td>
<td>4219</td>
<td>0</td>
<td>4163.45</td>
<td>0</td>
</tr>
<tr>
<td>Remote Write BW [MBytes/s]</td>
<td>1706.19</td>
<td>0</td>
<td>1705.09</td>
<td>0</td>
</tr>
<tr>
<td>Remote BW [MBytes/s]</td>
<td>5925.19</td>
<td>0</td>
<td>5868.54</td>
<td>0</td>
</tr>
</tbody>
</table>

Only one memory BW per socket (“Uncore”)

Half of BW comes from other socket!
If all fails…

- Even if all placement rules have been carefully observed, you may still see nonlocal memory traffic. Reasons?
  - Program has erratic access patterns → may still achieve some access parallelism (see later)
  - OS has filled memory with buffer cache data:

```
# numactl --hardware    # idle node!
available: 2 nodes (0-1)
node 0 size: 2047 MB
node 0 free: 906 MB
node 1 size: 1935 MB
node 1 free: 1798 MB
```

```
top - 14:18:25 up 92 days, 6:07, 2 users, load average: 0.00, 0.02, 0.00
Mem: 4065564k total, 1149400k used, 2716164k free, 43388k buffers
Swap: 2104504k total, 2656k used, 2101848k free, 1038412k cached
```
ccNUMA problems beyond first touch: Buffer cache

- **OS uses part of main memory for disk buffer (FS) cache**
  - If FS cache fills part of memory, apps will probably allocate from foreign domains
  - → non-local access!
  - “sync” is not sufficient to drop buffer cache blocks

- **Remedies**
  - Drop FS cache pages after user job has run (admin’s job)
    - seems to be automatic after aprun has finished on Crays
  - User can run “sweeper” code that allocates and touches all physical memory before starting the real application
  - **numactl** tool or **aprun** can force local allocation (where applicable)
  - Linux: There is no way to limit the buffer cache size in standard kernels
ccNUMA problems beyond first touch:
Buffer cache

Real-world example: ccNUMA and the Linux buffer cache

Benchmark:
1. Write a file of some size from LD0 to disk
2. Perform bandwidth benchmark using all cores in LD0 and maximum memory installed in LD0

Result: By default, Buffer cache is given priority over local page placement → restrict to local domain if possible!

(c) RRZE 2017
Back to the spMVM code: a little riddle

DLR1 matrix on 2x 8-core Sandy Bridge node

parallel first touch init.  serial init.

 GF/s

ccNUMA domain boundary

cores

GF/s

ccNUMA domain boundary

cores

(c) RRZE 2017
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>`

![Graph showing performance comparison between parallel placement, LD0 placement, and interleaved placement.](image)
Summary on ccNUMA issues

- **Identify the problem**
  - Is ccNUMA an issue in your code?
  - Simple test: run with `numactl --interleave`

- **Apply first-touch placement**
  - Look at initialization loops
  - Consider loop lengths and static scheduling
  - C++ and global/static objects may require special care
  - Depending on code and system, automatic page migration may do the trick

- **If dynamic scheduling cannot be avoided**
  - Distribute the data anyway, just do not use sequential placement!

- **OS file buffer cache may impact proper placement**
OpenMP performance issues on multicore

Barrier synchronization overhead
Topology dependence
The OpenMP-parallel vector triad benchmark

OpenMP work sharing in the benchmark loop

double precision, dimension(,:), allocatable :: A,B,C,D

allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
!$OMP PARALLEL private(i,j)
do j=1,NITER
!$OMP DO
 do i=1,N
    A(i) = B(i) + C(i) * D(i)
 enddo
!$OMP END DO
!$OMP END PARALLEL

(c) RRZE 2017
OpenMP vector triad on Sandy Bridge sockets (3 GHz)

- Performance [GFlop/s] vs Loop length
- Sync overhead grows with # of threads
- Bandwidth scalability across memory interfaces

Pattern: OpenMP barrier overhead

L1 core limit
Welcome to the multi-/many-core era

*Synchronization of threads may be expensive!*

```c
!$OMP PARALLEL ...
...
!$OMP BARRIER
!$OMP DO
...
!$OMP ENDDO
!$OMP END PARALLEL
```

Threads are synchronized at **explicit** AND **implicit** barriers. These are a main source of overhead in OpenMP programs.

Determine costs via modified OpenMP Microbenchmarks testcase (epcc)

On x86 systems there is no hardware support for synchronization!

- Next slides: Test OpenMP Barrier performance...
- for different compilers
- and different topologies:
  - shared cache
  - shared socket
  - between sockets
- and different thread counts
  - 2 threads
  - full domain (chip, socket, node)
## Thread Synchronization Overhead on IvyBridge-EP

*Barrier overhead in CPU cycles*

<table>
<thead>
<tr>
<th>2 Threads</th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared L3</td>
<td>599</td>
<td>425</td>
</tr>
<tr>
<td>SMT threads</td>
<td>612</td>
<td>423</td>
</tr>
<tr>
<td>Other socket</td>
<td>1486</td>
<td>1067</td>
</tr>
</tbody>
</table>

- **Full domain**
  - Socket (10 cores) | Intel 16.0: 1934 | GCC 5.3.0: 1301 |
  - Node (20 cores)  | Intel 16.0: 4999 | GCC 5.3.0: 7783 |
  - Node +SMT       | Intel 16.0: 5981 | GCC 5.3.0: 9897 |

- **2.2 GHz**
  - 10 cores
  - 10 cores

- **Strong topology dependence!**

- **Overhead grows with thread count**

- **Strong dependence on compiler, CPU and system environment!**
- **OMP_WAIT_POLICY=ACTIVE** can make a big difference
Thread synchronization overhead on Xeon Phi 7210 (64-core)
Barrier overhead in CPU cycles (Intel C compiler 16.03)

2 threads on distinct cores: 730

<table>
<thead>
<tr>
<th></th>
<th>SMT1</th>
<th>SMT2</th>
<th>SMT3</th>
<th>SMT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>One core</td>
<td>n/a</td>
<td>963</td>
<td>1580</td>
<td>2240</td>
</tr>
<tr>
<td>Full chip</td>
<td>5720</td>
<td>8100</td>
<td>9900</td>
<td>11400</td>
</tr>
</tbody>
</table>

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

3.2x cores (20 vs 64) on Phi
4x more operations per cycle per core on Phi

→ 4 · 3.2 = 12.8x more work done on Xeon Phi per cycle

1.9x more barrier penalty (cycles) on Phi (11400 vs. 6000)

→ One barrier causes 1.9 · 12.8 ≈ 24x more pain 😊.
Pattern-driven Performance Engineering

Basics of Benchmarking
Performance Patterns
Signatures
Basics of optimization

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 for benchmarking

- **Preparation**
  - Reliable timing (minimum time which can be measured?)
  - Document code generation (flags, compiler version)
  - Get access to an exclusive system
  - System state (clock speed, turbo mode, memory, caches)
  - Consider to automate runs with a script (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
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!
Performance Engineering Process: Analysis

Step 1 **Analysis**: Understanding observed performance

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

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

**Performance patterns** are typical performance limiting motifs
Performance Engineering Process: Modeling

Step 2 **Formulate Model:** Validate pattern and get quantitative insight

- **Pattern**
- **Qualitative view**
- **Performance Model**
- **Quantitative view**
- **Validation**
- **Traces/HW metrics**

*Optional* May be skipped!

(c) RRZE 2017
Performance Engineering Process: Optimization

Step 3 **Optimization**: Improve utilization of available resources

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

Performance improves until next bottleneck is hit
1. **Maximum resource utilization**  
   (computing at a bottleneck)

2. **Hazards**  
   (something “goes wrong”)

3. **Work related**  
   (too much work or too inefficiently done)
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Performance behavior</th>
<th>Metric signature, LIKWID performance group(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth saturation</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>ALU saturation</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>Inefficient data access</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>Micro-architectural anomalies</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>False sharing of cache lines</td>
<td>Large discrepancy from performance model in parallel case, bad scalability</td>
<td>Frequent (remote) CL evicts (CACHE)</td>
</tr>
<tr>
<td>Bad ccNUMA page placement</td>
<td>No parallel initialization</td>
<td>Unbalanced bandwidth on memory interfaces / High remote traffic (MEM)</td>
</tr>
<tr>
<td>Pipelining issues</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>Control flow issues</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</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>fraction</td>
<td>SpMVM scaling</td>
<td></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>Expensive instructions</td>
<td>Many cycles per instruction (CPI) if the problem is large-latency arithmetic</td>
</tr>
<tr>
<td>Expensive instructions</td>
<td>Similar to instruction overhead</td>
<td>Scalar instructions dominating in data-parallel loops (FLOPS_* , CPI)</td>
</tr>
<tr>
<td>Ineffective instructions</td>
<td>C/C++ aliasing problem</td>
<td></td>
</tr>
</tbody>
</table>

(c) RRZE 2017

Node-Level Performance Engineering

208
Patterns conclusion

- **Pattern signature** = performance behavior + hardware metrics

- Patterns are applies hotspot (loop) by hotspot

- Patterns map to **typical execution bottlenecks**

- Patterns are extremely helpful in classifying performance issues
  - The first pattern is always a hypothesis
  - Validation by tanking data (more performance behavior, HW metrics)
  - Refinement or change of pattern

- **Performance models** are crucial for most patterns
  - Model follows from pattern
Tutorial conclusion

- **Multicore architecture == multiple complexities**
  - Affinity matters → pinning/binding is essential
  - Bandwidth bottlenecks → inefficiency is often made on the chip level
  - Topology dependence of performance features → know your hardware!

- **Put cores to good use**
  - Bandwidth bottlenecks → surplus cores → functional parallelism!? 
  - Shared caches → fast communication/synchronization → better implementations/algorithms?

- **Simple modeling techniques and patterns help us**
  - ... understand the limits of our code on the given hardware
  - ... identify optimization opportunities
  - ... learn more, especially when they do not work!

- **Simple tools get you 95% of the way**
  - e.g., with the LIKWID tool suite
THANK YOU.
Presenter Biographies

Georg Hager holds a PhD in computational physics from the University of Greifswald. He is a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. His textbook “Introduction to High Performance Computing for Scientists and Engineers” is required or recommended reading in many HPC-related courses around the world. See his blog at http://blogs.fau.de/hager for current activities, publications, and talks.

Jan Eitzinger (formerly Treibig) holds a PhD in Computer Science from the University of Erlangen. He is now a postdoctoral researcher in the HPC Services group at Erlangen Regional Computing Center (RRZE). His current research revolves around architecture-specific and low-level optimization for current processor architectures, performance modeling on processor and system levels, and programming tools. He is the developer of LIKWID, a collection of lightweight performance tools. In his daily work he is involved in all aspects of user support in High Performance Computing: training, code parallelization, profiling and optimization, and the evaluation of novel computer architectures.

Gerhard Wellein holds a PhD in solid state physics from the University of Bayreuth and is a professor at the Department for Computer Science at the University of Erlangen. He leads the HPC group at Erlangen Regional Computing Center (RRZE) and has more than ten years of experience in teaching HPC techniques to students and scientists from computational science and engineering programs. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.
Abstract

- **SC17 tutorial:** Node-Level Performance Engineering
- **Presenter(s):** Georg Hager, Gerhard Wellein

**ABSTRACT:**

The advent of multi- and manycore chips has led to a further opening of the gap between peak and application performance for many scientific codes. This trend is accelerating as we move from petascale to exascale. Paradoxically, bad node-level performance helps to “efficiently” scale to massive parallelism, but at the price of increased overall time to solution. If the user cares about time to solution on any scale, optimal performance on the node level is often the key factor. We convey the architectural features of current processor chips, multiprocessor nodes, and accelerators, as far as they are relevant for the practitioner. Peculiarities like SIMD vectorization, shared vs. separate caches, bandwidth bottlenecks, and ccNUMA characteristics are introduced, and the influence of system topology and affinity on the performance of typical parallel programming constructs is demonstrated. *Performance engineering* and *performance patterns* are suggested as powerful tools that help the user understand the bottlenecks at hand and to assess the impact of possible code optimizations. A cornerstone of these concepts is the roofline model, which is described in detail, including useful case studies, limits of its applicability, and possible refinements.
Selected references

Book:

Papers:
References

References


References


(c) RRZE 2017 Node-Level Performance Engineering
References
