Detecting performance limiting factors with hardware monitoring

Thomas Gruber, HPC group, FAU Erlangen
Agenda

- What is performance?
- What are limiting factors?
- Introduction to LIKWID
- Detection
  - Bad vectorization
  - Load imbalance
  - False sharing in cache hierarchy
Motivation
Motivation

HPC == Computing on the edge

- Execution limited by bottlenecks caused by
  - Hardware system limitations
  - Software runtime behavior
  - User code
- (Best) Approach in HPC:
  - Keep user code slim (main logic, glue code & library calls)
  - Use optimized software runtime and libraries
  - Hit hardware bottleneck
What is performance?
How to measure performance?
Performance

What is a good measure/metric?

```c
for(i=0; i<n; i++) {
    a[i]= 3.0*c0+c1*c2+c3*c4*a[i]-1.0*d0*a[i];
}
```

\#FLOPS = 8 * n

---

Same execution time but...

```
\#FLOP = 2 * n + 5
```

d0 = 3.0*c0+c1*c2;
d1 = c3*c4-1.0*d0;

```c
for(i=0; i<n; i++) {
    a[i]= d0 + d1*a[i];
}
```

... but my MFlop/s rate is only \frac{1}{4}!

My vector update code runs at 2,000 MFLOP/s on a 2GHz processor!
Performance

What is a good measure/metric?

- **Performance** = WORK / TIME
- “Pure” metrics – basic choices for “WORK”
  - **Iterations**: Total number of loop iterations performed
  - **MFLOP**: Millions of Floating Point Operations
  - **MIPS**: Millions of Instructions
  - **Lattice Site/Cell/Particle Updates**: number of sites/cells/particles to be updated/computed
  - **Physical simulation time**: Physical time (e.g. nanoseconds) a system is propagated
Performance

What is a good measure/metric?

- Simplest performance metric ("Bestseller"): $1 / \text{TIME}$
  - Measures time to solution
  - Carefully specify the "problem" you solved!
  - Best metric thinkable, but not intuitive in all

- LINUX / UNIX command `time`

```plaintext
> time ./test.x
> 34.650u 0.612s 0:35.28 99.9%
> time ./testwIO.x
> 33.802u 0.608s 0:43.64 78.8%
```

- CPU time – it’s evil!
- No I/O!

Problem: Which TIME?

Always use one dedicated source for time measurements (e.g. one node)

This is the time (walltime) you wait for the result!
Performance Measurement

**Best Practices**

- **Preparation**
  - Reliable timing/timer granularity
    (Minimum time which can be measured?)
  - Document code generation (Flags, Compiler Version)
  - Document system state (Clock, Turbo mode, Memory, Caches)
  - Consider to automate runs with a script (Shell, python, perl)
    if it works manually
Performance Measurement

Best Practices

- Doing it
  - Get exclusive system
  - Fix clock speed
  - Control Affinity / Topology
    Where does my code/threads/processes run exactly?
  - Working set size – code input parameters?!
  - Is result deterministic, reproducible and reasonable

→ Do statistics (Mean, Best, …)
Performance Measurement

Best Practices

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

- Is there a (simple) model which can (qualitatively) explain the performance levels and variations?

```
    do k = 1 , Nk; do j = 1, Nj
      do i = 1, Ni
        y(i,j,k) = const*
        $ ( 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
```

Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz
Memory Bandwidth 48 GB/s
What are limiting factors?
Performance

Impact factors

- For a given code/problem performance may be influenced by many factors

- For reproducibility of results all performance critical factors need to be reported!
- Sensibility and stability analysis!
- Statistics - fluctuations between runs

CPU
- Clock speed, SMT, #cores, cache size

Memory
- interface, size, speed

Vendor / Board

I/O subsystem

BIOS Settings

Compiler
- Version, Flags

Libraries
- gnu, Intel, pgi, pathscale
- Atlas, mkl, fftw,…

OS
- Parameters, Version, Libraries

SuSe, RedHat, Ubuntu,…
What is hardware performance monitoring?
What is LIKWID?
What is hardware performance monitoring? What about software performance monitoring?

- People write code and don’t design hardware
- Execution behavior is specified by my code
- Applications are just compiled code. Compilers are always right

- “I know how much data is consumed, it’s my code!”
- “I already sum up the number of FLOPs!”
- “Of course I overloaded the operators, it’s C++”
What is hardware performance monitoring?

Overview about HPM

- Performance monitoring units (PMUs) at hardware level
- Introduced for x86 with Intel Pentium (1994)
- Originally used by CPU vendors for hardware validation
- No additional CPU work to process hardware events in PMUs
- Accessing PMUs requires CPU work → Overhead
- Limited number of counters per PMU
Tools for hardware performance monitoring

- Kernel-Interface `perf_event`
  - Kernel handles management of PMUs (wrong kernel = no support)
  - Counting per OS processes or HW threads
  - Feature-rich interface used by many tools: `perf`, PAPI, ...

- LIKWID
  - Runs in user-space. Uses common kernel interfaces or `perf_event`
  - Counts per HW thread (knowledge about processes)
  - Derives metrics out of raw counter values

Of course: Intel VTune
Overview

- "Like I Knew What I’m Doing"
- Tool suite for performance-oriented programmers
- Developed by HPC group of RRZE since 2009
- Open source and we are working on it

Goals:
- Provide support for architecture at official release date
- Offer powerful tools for daily work
How to use LIKWID on RWTH systems

- CLAIX18
  - srun --reservation=aixcel_perf \ 
    --pty /usr/local_rwth/bin/zsh
  - module load likwid
  - Bash currently not working

- CLAIX16
  - #BSUB -app likwid
  - module load likwid
**LIKWID**

*System topology*

*likwid-topology*

- System information
- Thread, cache and NUMA topology

**Process affinity**

*likwid-pin -c <cpustr> ./a.out*

- Handles common threading solutions
- Simple cpu selection syntax
- *likwid-pin -c S0:0-9@S1:0-9 ./a.out*

10 processes on socket0 and socket1 (phys. cores first)
LIKWID
HPM with *likwid-perfctr*

- Control application for HPM measurements
- Measurement modes:
  - Start-to-end
  - Timeline (time-based sampling)
  - Stethoscope
  - MarkerAPI (code instrumentation)
  - likwid-perfctr –c/–C <cpusel> -g <eventlist> (opts) ./app
  - -c: Measure on cores in cpusel, -C: Measure & pin
  - -g: List of event+counter combis or performance group
  - List of all counters and events: likwid-perfctr –e
  - Search for some event: likwid-perfctr –E <searchstr>
$ likwid-perfctr -C 0,1 -g L2_TRANS_L1D_WB:PMC0 ./app

+---------------------------------------------+---------------------------------------------+
| Event                                      | Counter | Core 0 | Core 1 |
+---------------------------------------------+---------------------------------------------+
| Runtime (RDTSC) [s]                         | TSC     | 2.573182e+00 | 2.573182e+00 |
| L2_TRANS_L1D_WB                             | PMC0    | 281176518   | 281240170   |
+---------------------------------------------+---------------------------------------------+

- Event names (in many cases) not intuitive
- Events are architecture-specific
- Some sound promising but return bad counts, others are broken
- More interest in real metrics like volume of loaded data
LIKWID

HPM with **likwid-perfctr**

- LIKWID defines performance groups
  \[\approx\] eventlist + derived metrics + documentation
- List all groups: likwid-perfctr -a

```bash
$ likwid-perfctr -C 0,1 -g L2 ./app
```

<table>
<thead>
<tr>
<th>Metric</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s]</td>
<td>2.6439</td>
<td>2.6439</td>
</tr>
<tr>
<td>L2D write bandwidth [MBytes/s]</td>
<td>6744.8121</td>
<td>6743.6037</td>
</tr>
<tr>
<td>L2D write data volume [GBytes]</td>
<td>17.8325</td>
<td>17.8293</td>
</tr>
<tr>
<td>&lt;there is more&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

04.12.2018 | Detecting performance limiting factors with hardware monitoring | Thomas Gruber
Find hot functions (if you don’t know)

- gprof, Intel compiler, xray (clang) or perf record

```bash
gcc -pg ... && ./a.out && gprof <opts> a.out gmon.out
```

Get a first impression where the time is spent

Get hot functions (if you don’t know)

- gprof, Intel compiler, xray (clang) or perf record

```bash
gcc -pg ... && ./a.out && gprof <opts> a.out gmon.out
```

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time seconds</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>95.70</td>
<td>12.47</td>
<td>12.47</td>
</tr>
<tr>
<td>3.45</td>
<td>12.92</td>
<td>0.45</td>
</tr>
<tr>
<td>0.61</td>
<td>13.00</td>
<td>0.08</td>
</tr>
<tr>
<td>0.08</td>
<td>13.03</td>
<td>0.01</td>
</tr>
<tr>
<td>0.00</td>
<td>13.03</td>
<td>0.00</td>
</tr>
</tbody>
</table>

And now?

04.12.2018 | Detecting performance limiting factors with hardware monitoring | Thomas Gruber
LIKWID

HPM of functions

- Code instrumentation using LIKWID’s MarkerAPI

```c
#include <likwid.h>
LIKWID_MARKER_INIT; // in serial region
LIKWID_MARKER_REGISTER("Compute"); // in parallel region
LIKWID_MARKER_START("Compute");
<code>
LIKWID_MARKER_STOP("Compute");
LIKWID_MARKER_CLOSE; // in serial region
```

- Compile with `-DLIKWID_PERFMON`
- `likwid-perfctr -C 0,1 -g L2 -m ./app`

Recommended: Reduces startup overhead

Multiple regions and nesting allowed

Activate MarkerAPI mode

Also available for FORTRAN
## LIKWID

### HPM of functions

```bash
$ likwid-perfctr -C 0,1 -g L2 -m ./app 2000000
```

Region **Compute**, Group 1: L2

<table>
<thead>
<tr>
<th>Region Info</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDTSC Runtime [s]</td>
<td>77.724980</td>
<td>77.725490</td>
</tr>
<tr>
<td>call count</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

[...]

<table>
<thead>
<tr>
<th>Metric</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s]</td>
<td>77.7250</td>
<td>77.7255</td>
</tr>
<tr>
<td>L2 bandwidth [MBytes/s]</td>
<td>10275.9891</td>
<td>10272.4909</td>
</tr>
<tr>
<td>L2 data volume [GBytes]</td>
<td>798.7010</td>
<td>798.4344</td>
</tr>
</tbody>
</table>

If we know that each region executes 2M iterations:

\[
\begin{align*}
1597.135 \text{ GByte} \quad &\quad 1E3 \times 2E6 \text{ iterations} \\
\approx 798 \text{ Byte} \quad &\quad \text{iterations}
\end{align*}
\]

If we know that each region executes 2M iterations:
LIKWID
HPM + MPI with likwid-mpirun

- Wrapper for MPI for pinning and HPM
  
  \$ likwid-mpirun -nperdomain S:1 ./a.out
  
  One process per socket on all hosts
  
  \$ likwid-mpirun -pin S0:0-3_S1:0-3 ./a.out
  
  Two processes per node, each using 4 threads
  
  \$ likwid-mpirun -np \textless p\textgreater{} -g FLOPS_DP -m ./a.out
  
  Measure DP Flops on with all procs (with MarkerAPI)

- Works with Intel MPI, OpenMP, Mvapich and SLURM

\texttt{slalloc -N X; likwid-mpirun -np Y ./a.out}

At CLAIX18 use:

--mpi intelmpi
Detection of limiting factors

- Bad vectorization
- Load imbalance
- False sharing in cache hierarchy
Hands on – Bad vectorization

- Modern CPUs gain most of their FP power from vectorization
- Intel compiler does a good job detecting vectorizable code
  - GCC not that good, use –f ##################################################################
  - m[avx2|avx512f]
- Mark loops with proper pragmas:
  - #pragma simd
  - #pragma vector align
- Specify build target with -x<target> like -xCORE-AVX512 for Skylake (-xHost sometimes doesn’t do the job!)
  (Force AVX512: -qopt-zmm-usage=high)
Hands on – Bad vectorization

Performance Comparison with matrix-vector-multiplication

You don’t want to miss this gain!

This increase can be larger for other codes

Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
Hands on – Bad vectorization

- Detect with FLOPS_DP or FLOPS_SP groups

<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>DP MFLOP/s</td>
<td>2204.7856</td>
<td>2214.7243</td>
<td>2213.3518</td>
<td>2209.7715</td>
</tr>
<tr>
<td>AVX DP MFLOP/s</td>
<td>2203.4644</td>
<td>2213.3972</td>
<td>2212.0254</td>
<td>2208.4474</td>
</tr>
<tr>
<td>Packed MUOPS/s</td>
<td>550.9762</td>
<td>553.4599</td>
<td>553.1168</td>
<td>552.2222</td>
</tr>
<tr>
<td>Scalar MUOPS/s</td>
<td>1.1011</td>
<td>1.1060</td>
<td>1.1054</td>
<td>1.1036</td>
</tr>
</tbody>
</table>

- Problem: Hardware counts instructions, AVX512 supports masking → Reported numbers might be too high

Almost all FP ops are AVX
Detection of limiting factors

- Load imbalance
Dense matrix-vector multiplication in DP
Parallelization

- Let’s assume symmetric matrix
  - Avoid unneeded computation by calculating only triangular matrix
  - Distribute columns to threads
- Problem?
  Can you name it?

```c
!$OMP PARALLEL DO PRIVATE(r)
do  c = 1 , SIZE
  tmp = x(c)
do  r = 1 , c
    y(r) = y(r) + A(r,c)* tmp
  enddo
enddo
!$OMP END PARALLEL DO
```
Dense matrix-vector multiplication in DP

Parallelization

$ likwid-perfctr -C 0,1,2 -g L2 -m ./a.out

CPU type:       Intel Core SandyBridge EN/EP processor
CPU clock:      3.09 GHz

Group 1: Region Compute

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</th>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>FIXC0</td>
<td>2.626800e+08</td>
<td>3.187585e+08</td>
<td>3.780255e+08</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>FIXC1</td>
<td>4.972802e+08</td>
<td>4.961411e+08</td>
<td>4.933711e+08</td>
</tr>
<tr>
<td>CPU CLK UNHALTED REF</td>
<td>FIXC2</td>
<td>4.972801e+08</td>
<td>4.961404e+08</td>
<td>4.933714e+08</td>
</tr>
<tr>
<td>L1D REPLACEMENT</td>
<td>PMC0</td>
<td>5.490278e+07</td>
<td>3.927353e+07</td>
<td>2.364295e+07</td>
</tr>
<tr>
<td>L1D M EVICT</td>
<td>PMC1</td>
<td>2.920200e+04</td>
<td>2.876600e+04</td>
<td>2.861000e+04</td>
</tr>
</tbody>
</table>

Always use a reasonable metric for “work”!

Number of instructions raise with CPU number

First CPU loads most data into L1

Same amount of CLs evicted from L1
Dense matrix-vector multiplication in DP

- Here we look at 256 bit wide vector operations → AVX
- First thread “works” most
Dense matrix-vector multiplication in DP

Parallelization

- How to balance the workload for every thread?
- Over-engineering:
  - Different data layout
  - Use smaller chunks of work (split rows/columns in multiple chunks?)
- Simple: Change OpenMP schedule `schedule(kind, chunksize)`
  - dynamic: actively balance chunks of work → Overhead
  - static: pre-compute chunks of work → No Overhead 😊
  - chunksize parameter describes size of chunks
Dense matrix-vector multiplication in DP

Parallelization

- Reduce chunk of work to a few rows
- Not optimal, but sufficient!

- Parallelization
  - Saturate memory bandwidth

![Graph showing vectorized double precision operations and floating point operations over number of threads for different OMP schedules and SIMD FP 256 packed double operations.](image)
Dense matrix-vector multiplication in DP

Parallelization

- `malloc()` does not locate data
- First access defines location (first touch)
- Data locality crucial for high performance on ccNUMA systems

Simplification: Use same work sharing for allocation and work!

```
numactl -i <nodesel> ./a.out
```

Not always, but true for GCC and Intel/LLVM OpenMP: Once started and pinned threads stay there!
Detection of limiting factors

- False-sharing
Hands on – False-sharing

- Serial histogram code
  ```c
  int hist[8] = { 0 };
  for(j=0; j<1000000000; ++j) {
      hist[rand_r(&seed) & 0x7]++;
  }
  ```

  Possible data race!
  (if run in parallel)
Hands on – False-sharing

Parallelize histogram code

- Solution: Own histogram per thread

```c
int hist[8][MAX_THREADS] = { 0 };  
#pragma omp parallel for firstprivate(seed) 
for(j=0; j<1000000000; ++j) 
    hist[rand_r(&seed) & 0x7][tid]++; 
// combine thread histograms
```

No data race! But….
Hands on – False-sharing

Parallelize histogram code

![Diagram showing cache levels and false-sharing between cores C0 and C1, with 64B cache line.]
Hands on – False-sharing

Parallelize histogram code

Single thread:
4.85 s
10 threads:
7.58 s

After Optimization with thread-local histogram
10 threads: 0.61 s

Intel IvyBridge EP
Hands on – False-sharing

Problems

- CL sharing is normal in almost every code
  - Global data structures (stop criterion, memory addresses, …)
  - Common structs (static information)
- Cannot differentiate between required shared CL and falsely shared CL
- Counters not accurate (Haswell: up to 50% deviation)
- No information about cache flushes / memory barriers
# Useful performance groups

<table>
<thead>
<tr>
<th>Topic</th>
<th>Perf. group</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP operations</td>
<td>FLOPS_DP FLOPS_SP</td>
</tr>
<tr>
<td>Memory traffic</td>
<td>MEM</td>
</tr>
<tr>
<td>Quality of cache usage</td>
<td>L2CACHE L3CACHE</td>
</tr>
<tr>
<td>Energy usage</td>
<td>ENERGY</td>
</tr>
<tr>
<td>TLBs</td>
<td>TLB_DATA TLB_INSTR</td>
</tr>
<tr>
<td>Branch prediction</td>
<td>BRANCH</td>
</tr>
<tr>
<td>In-core stalls</td>
<td>CYCLE_ACTIVITY</td>
</tr>
</tbody>
</table>
Thank you for your attention!
Regionales RechenZentrum Erlangen [RRZE]
Martensstraße 1, 91058 Erlangen
http://www.rrze.fau.de

https://github.com/RRZE-HPC/likwid/wiki
Detection of limiting factors

- Bandwidth limitation
Motivation

A Short look back

Latency as memory technology has matured
- Main limitation nowadays is memory bandwidth/latency
- How to detect code is limited by memory bandwidth?
  - You need to know how much data you need to load/store from/to memory
  - Use a model like Roofline or ECM (more insight but more work)
  - Data for Roofline: MEM_DP, MEM_SP

Clearly memory bound

And now? Not clear at all
Jacobi 2D 5pt stencil (DP)

Data traffic 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
```

Available in cache (used 2 iterations before)

LD+ST y(j,k)
(incl. write allocate)

LD x(j+1,k)

LD x(j,k+1)

LD x(j,k-1)

Naive balance (incl. write allocate):

\[
\begin{align*}
x( :, :) & : 3 \text{ LD +} \\
y( :, :) & : 1 \text{ ST+ 1LD}
\end{align*}
\]

\[\Rightarrow B_C = 5 \text{ Words} / \text{ LUP} \Rightarrow B_C = 40 \text{ B} / \text{ LUP} \]

(assuming double precision)

Reminder “write allocate”:
If a cache line is not present in L1 cache at a store, the CL needs to loaded first into L1.
Questions:

1. How to achieve 24 B/LUP also for large $j_{max}$?
2. How to sustain > 600 MLUP/s for $j_{max} > 10^4$?
3. Why 24 B/LUP anyway??

Naïve balance was 40B/LUP!

Jacobi 2D 5pt stencil (DP)

Data traffic analysis

Code balance ($B_c$) measured with LIKWID

$\frac{j_{max} = k_{max}}{}$ $\geq \frac{j_{max} \times k_{max}}{\text{const}}$

Intel Xeon E5-2690 v2 @ 3 GHz, ifort V13.1, L1: 32kB, L2: 256kB, L3: 25 MB
Jacobi 2D 5pt stencil (DP)

Data traffic analysis

Reduce inner (j-) loop dimension successively

Best case: 3 „layers“ of grid fit into the cache!
Jacobi 2D 5pt stencil (DP)

Data traffic 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
```

```
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
```

3 * jmax * 8B < CacheSize/2
"Layer condition" fulfilled?

- YES
- NO

y: (1 LD + 1 ST) / LUP
x: 1 LD / LUP

B_C = 24 B / LUP

y: (1 LD + 1 ST) / LUP
x: 3 LD / LUP

B_C = 40 B / LUP
Jacobi 2D 5pt stencil (DP)
Establish Layer condition for all $j_{\text{max}}$

- Apply spatial blocking of j-loop

```fortran
  do $jb=1,j_{\text{max}},j_{\text{block}}$ ! Assume $j_{\text{max}}$ is multiple of $j_{\text{block}}$
    do $k=1,k_{\text{max}}$
      do $j=jb,(jb+j_{\text{block}}-1)$ ! Length of inner loop: $j_{\text{block}}$
        $y(j,k) = \text{const} \times (x(j-1,k) + x(j+1,k) \& + x(j,k-1) + x(j,k+1))$
      enddo
    enddo
  enddo
```

- Determine for given CacheSize an appropriate $j_{\text{block}}$:

  New layer condition (blocking)

  $3 \times j_{\text{block}} \times 8B < \text{CacheSize}/2 \quad \Rightarrow \quad j_{\text{block}} < \text{CacheSize} / 48 \text{ B}$
Jacobi 5pt stencil (DP)

Blocking for which cache?

Intel Xeon E5-2690 v2 @ 3 GHz, ifort V13.1, L1: 32kB, L2: 256kB, L3: 25 MB

Blocking factor (CS=25MB) too large!

L2: CS=256 KB
\[ j_{\text{block}} = \min(j_{\text{max}}, 5333) \]

L3: CS=25 MB
\[ j_{\text{block}} = \min(j_{\text{max}}, 533333) \]

Where does the gap come from?

Main Memory

Code balance (B_C) measured

\[ j_{\text{max}} = k_{\text{max}} \]
\[ j_{\text{max}} \times k_{\text{max}} = \text{const} \]

Little overhead at block boundaries

04.12.2018 | Detecting performance limiting factors with hardware monitoring | Thomas Gruber
Jacobi 2D 5pt stencil (DP)

What about 3D 7pt stencil?

3*jmax * 8B < CacheSize/2

```plaintext
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

do k=1,kmax
  do j=1,jmax
    do i=1,imax
       y(i,j,k) = const * (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
```

3*jmax * imax * 8B < CacheSize/2

B_c = 24 B / LUP

“Layer condition” OK → 5 accesses to x() served by cache

Same for version with spatial blocking

04.12.2018 | Detecting performance limiting factors with hardware monitoring | Thomas Gruber
Jacobi 3D 7pt stencil (DP)

**Blocking, layer condition and threading?**

```
&!

do jb=1,jmax,jblock ! Assume jmax is multiple of jblock

  !$OMP PARALLEL DO SCHEDULE(STATIC)
  do k=1,kmax

    do j=jb, (jb+jblock-1) ! Loop length jblock
      do i=1,imax

        y(i,j,k) = const *(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

enddo

!$OMP END PARALLEL DO
```

"Layer condition" (j-Blocking))

\[ \text{nthreads} \times 3 \times j\text{block} \times \text{imax} \times 8\text{B} < \text{CS}/2 \]

\[ j\text{block} < \text{CS}/(\text{imax} \times \text{nthreads} \times 48\text{B}) \]

**Intel Xeon E5-2690 v2 @ 3 GHz, ifort V13.1, L1: 32kB, L2: 256kB, L3: 25 MB**

**Maximum memory bandwidth** \( b_S = 48 \text{ GB/s} \)

**Best performance:**

\[ \frac{b_S}{B_C} = 2000 \text{ MLUP/s} \]