

Introduction to Performance Engineering on HPC

Holger Obermaier | 22. June 2022

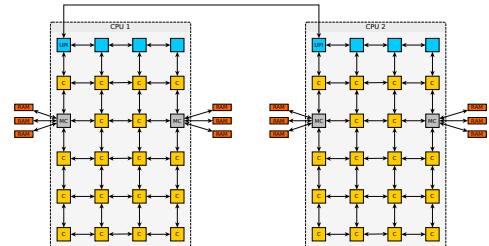
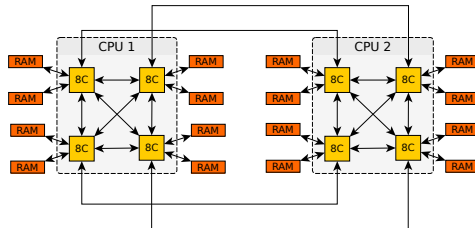


1. Optimization cycle
2. Tool Test Cases
3. Likwid Tools: Overview
4. Likwid Tools: `likwid-topology`
5. Likwid Tools: `likwid-bench`
6. Compiler Optimization Report
7. `/usr/bin/time`
8. Application Performance Snapshot (APS)
9. Likwid Tools: `likwid-perfctr`
10. Likwid Tools: `likwid-perfctr` Marker API
11. `perf` tools
12. Intel Trace Analyzer and Collector (ITAC)
13. References

Optimization cycle

Current state of hardware development

- CPU cores do not get faster anymore
 - More and more cores and nodes
 - Multiple levels of caches try to hide memory latency
- ⇒ Optimizing code gets more complex
- ⇒ Support by performance tools is needed



Optimization cycle (2)

Iterative process

- Collect hardware information
- Collect performance data
- Analyze hardware information and performance data
 - Where is most of the time spent?
 - What is the expected performance?
 - Are cores evenly utilized?
 - Is memory access local?
 - Does communication limit performance?

Optimization cycle (3)

Iterative process (continued)

- Fix problem
 - Appropriate data structure (e.g. Array of structs vs. struct of arrays)
 - Loop layout (allow compiler vectorization, CPU prefetching)
 - Blocking (Cache reuse)
 - Compiler and MPI command line options (e.g. process binding)
- Repeat until effort is no longer worth expected improvement

This talk focuses on hardware information and performance data collection and analysis

Tool Test Cases

Benchmark *stream*

Copy $c = a$, $a, c \in \mathbb{R}^n$

Scale $b = \alpha c$, $b, c \in \mathbb{R}^n$, $\alpha \in \mathbb{R}$

Add $c = a + b$, $a, b, c \in \mathbb{R}^n$

Triad $a = b + \alpha c$, $a, b, c \in \mathbb{R}^n$, $\alpha \in \mathbb{R}$

- $\mathcal{O}(n)$ memory operations, $\mathcal{O}(n)$ compute operations

⇒ Memory bandwidth bound

Tool Test Cases

Benchmark *dgemm*

Multiply $C = A \cdot B$, $A, B, C \in \mathbb{R}^{n \times n}$

- $\mathcal{O}(n^2)$ memory operations, $\mathcal{O}(n^3)$ compute operations

⇒ Floating point bound

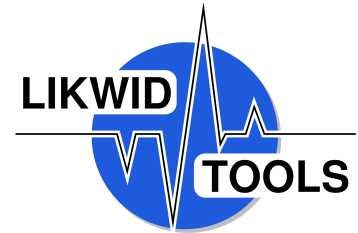
Benchmark *rank_league*

- Asynchronous point to point MPI communication
- $\mathcal{O}(1)$ memory operations, $\mathcal{O}(1)$ compute operations

⇒ Communication bound

Likwid Tools

- Collection of simple command line tools
- Hardware information:
`likwid-topology`
- Micro benchmarks:
`likwid-bench`
- Pinning:
`likwid-pin`, `likwid-mpirun`
- Performance counters:
`likwid-perfctr`



Likwid Tools: likwid-topology

- CPU topology (hardware threads, cores, sockets)
- Cache topology (location and size of caches)
- Cache properties (cache line size, associativity)
- NUMA topology (location and size of main memory)
- Get knowledge on how to bind your tasks, pin your threads

Example

- [likwid-topology on Intel Xeon Ice Lake](#)
- [likwid-topology cache topology on Intel Xeon Ice Lake](#)

Hands On

Preparation

- Get familiar with `likwid-topology`. Use
 - h to get help
 - g to get a graphical output
 - c to get cache information
- Be aware `uc1` and `uc1e` have different hardware.
- For the hands on examine the questions on the login node

Questions

- How many hardware threads, cores, sockets are available?
- How many cache levels are available?
- Which sizes do they offer?
- How many NUMA domains are available?

Likwid Tools: likwid-bench

What is the maximum

- achievable memory bandwidth
- achievable cache bandwidth
- achievable computing power
- Vector (AVX, AVX2, AVX-512) computing power
- Fused multiply-add (FMA) computing power

Example

- likwid-bench on Intel Xeon Ice Lake [!\[\]\(511a36c244659513b679df9c639945de_img.jpg\)](#)

Hands On

Preparation

- Start an interactive one node job
- Get familiar with `likwid-bench`. Use
 - h to get help
 - a to list available micro benchmarks
 - l to list properties of test
 - p to list available thread domains
- Use micro benchmarks `stream_avx_fma` and `stream_mem_avx_fma` to answer the questions

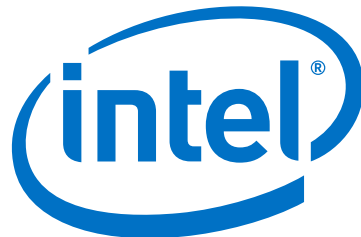
Questions

- What memory bandwidth can be reached using only one thread?
- What is the maximum achievable main memory bandwidth?
- What about L1, L2 and L3 cache bandwidth?


Compiler Vectorization Report (Intel)

- Usage vectorization report

```
module add compiler/intel/19.1  
icc ${OPT_FLAGS} \  
    -qopt-report \  
    -qopt-report-phase=vec \  
    -qopt-report-stdout \  
    ${SOURCE} -o ${OUTFILE}
```



Example

Intel vectorization report: [stream](#) 

Compiler Vectorization report (GCC)

- Usage vectorization report

```
module add compiler/gnu  
gcc ${OPT_FLAGS} \  
    -fopt-info-vec \  
    ${SOURCE} -o ${OUTFILE}
```



Example

GCC vectorization report: stream [↗](#)

Hands On

Preparation

- Change to folder HandsOn/Stream
- Use script `./build.intel_vec_report.sh` to generate Intel compiler vectorization report
- Use script `./build.gnu_opt_report.sh` to generate GCC compiler vectorization report

Questions


- Were Intel and GNU compiler able to vectorize the loops in the functions `tuned_STREAM_Copy`, `tuned_STREAM_Scale`, `tuned_STREAM_Add` and `tuned_STREAM_Triad`?
- Why is the loop in `tuned_STREAM_Scale` (line 337) mentioned twice in the Intel vectorization report?
- Why is no peel loop needed for the loop in `tuned_STREAM_Scale` (line 337)?

`/usr/bin/time`

- No recompilation needed
⇒ Use your existing binary
- Uses kernel resource usage info
- Report time consumption
 - time spent in user space
 - time spent in kernel space
 - elapsed time
- Report memory consumption
 - maximum resident size
 - Page faults
- Report IO operations



Example

Comparison *stream* serial/parallel execution with `time` 

Preparation

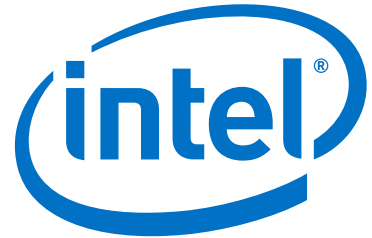
- Change to folder HandsOn/Stream
- Use script `./build.sh` to build stream benchmark
- Use `msub jobscript.time.msub` to submit batch job

Questions

- What is the difference between the two stream benchmark runs in `jobscript.time.msub`?
- Where can you see the difference in the output of `/usr/bin/time`?
- What causes the high amount of system time?
- Do memory consumption reported by stream benchmark and `/usr/bin/time` match?

Application Performance Snapshot (APS)

- No recompilation needed
⇒ Use your existing binary
- But: Best compatibility with Intel compiler and MPI
- Uses MPI library instrumentation
- Quick insight into
 - MPI
 - OpenMP
 - Memory access
 - Floating point
 - IO usage
- Text and HTML report







Application Performance Snapshot (APS) (2)

- Usage serial or OpenMP binary

```
module add compiler/intel/2022  
module add devel/vtune/2022  
aps ${BINARY}
```

Example



- APS: stream 
- APS HTML report: stream 
- APS: dgemm 
- APS HTML report: dgemm 

Application Performance Snapshot (3)

- Usage MPI binary

```
module add compiler/intel/2022 \  
          mpi/impi/2021.5.1  
module add devel/vtune/2022  
mpirun aps ${BINARY}
```

Example

- APS: rank_league 
- APS HTML report: rank_league 

Hands On

Preparation

- Change to folder HandsOn/Stream
- Use script `./build.sh` to build stream benchmark
- Use `msub jobscript.ap.s.msub` to submit batch job
- Repeat these steps in folder HandsOn/Dgemm and HandsOn/Rank_league

Questions

What are the limiting factors for benchmark

- stream?
- dgemm?
- rank_league?

Likwid Tools: likwid-perfctr

- Measures total program performance
- No recompilation needed \Rightarrow Use your existing binary
- Uses hardware performance *counters*
- Uses *sampling*
 - Low overhead
 - Only statistical results
- Performance groups simplify HW counters use
- Important performance groups
 - FLOPS_AVX Packed AVX MFLOP/s
 - MEM Main memory bandwidth
 - UPI Traffic on the UPI (socket interconnect)

Likwid Tools: likwid-perfctr (2)

■ Usage

```
likwid-perfctr -a # Available performance groups  
likwid-perfctr -H --group ${GROUP} # Group information  
likwid-perfctr --group ${GROUP} -C ${CPU_LIST} ${BINARY} # Measure
```

Example

- likwid-perfctr: Performance group MEM and UPI on benchmark stream ↗
- likwid-perfctr: Performance group FLOPS_AVX on benchmark dgemm ↗

Hands On

Preparation

- Get familiar with `likwid-perfctr`. Use
 - h to get help
 - a to list available performance groups
 - H to get performance group help (e.g. for group NUMA)
- Change to folder `HandsOn/Stream`
- Use script `./build.sh` to build stream benchmark
- Use `msub jobscript.perfctr.msub` to submit batch job

Questions

- What is the difference between the two stream benchmark runs in `jobscript.perfctr.msub`?
- Where can you see the difference in the output of stream benchmark
- Where can you see the difference in the output of `likwid-perfctr`?

Likwid Tools: likwid-perfctr Marker API

- Measure partial program performance
- Add likwid marker API to source code. Recompile.
- API macros:

`likwid_markerInit` Initialize likwid marker API



`likwid_markerThreadInit` Initialize each thread

`likwid_markerStartRegion` Start a measurement in named region

`likwid_markerStopRegion` Stop a measurement in named region

`likwid_markerClose` Close likwid marker API

Example

- Likwid marker API: stream 
- Likwid marker API: dgemm 

Hands On

Preparation

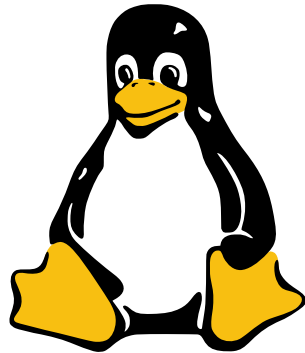
- Compare stream source code in folders HandsOn/Stream and HandsOn/Stream.likwid
- Change to folder HandsOn/Stream.likwid
- Use scripts ./build.gnu.sh and ./build.intel.sh to build stream benchmark
- Use `msub jobscript.gnu.msub` and `msub jobscript.intel.msub` to submit batch jobs

Questions

- Investigate region scale. Remember region scale should contain as many reads as write operations. Why is the read volume
 - twice as high as the write volume when using GNU compiler?
 - equal to write volume when using Intel compiler?

perf tools

- Part of Linux kernel
- No recompilation needed
⇒ Use your existing binary
- Uses hardware performance *counters*
- Uses *sampling*
 - Low overhead
 - Only statistical results
- Find *hot spots*
(functions or code regions)
- Record *call graph*
(with compiler flag `-g`)





perf tools (2)

■ Usage

```
perf list           # available HW counters
perf stat ${BINARY} # profile w. HW counters
perf record ${BINARY} # measurement -> perf.data
perf report         # Hot spot report
perf annotate        # Annotated assembler code
```

Example

- perf: dgemm 
- perf: stream 

Hands On

Preparation

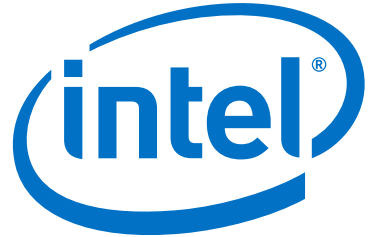
- Get familiar with `perf`
- Change to folder `HandsOn/Stream`
- Use scripts `./build.debug.sh` to build stream benchmark with debug symbols
- Use `msub jobscript.perf.msub` to submit batch job

Questions

- What are the 4 hot spots of `stream`?
- Navigate to `tuned_STREAM_Triad`
 - What assembler instructions are used?
 - Do they use vector registers?

Intel Trace Analyzer and Collector (ITAC)

- No recompilation needed
⇒ Use your existing binary
- Uses *sampling*
 - Low overhead
 - Only statistical results
- Uses MPI library instrumentation
 - Collect non-statistical data
 - *Communication pattern*
 - *Message sizes*
- Can use compiler instrumentation
 - Can cause significant overhead
 - Collect non-statistical data
 - *Call graph*




Intel Trace Analyzer and Collector (ITAC) (2)

- Graphical tool shows
 - Event timeline
 - Quantitative timeline
 - Function profile
 - Message profile

- Usage

```
module add devel/itac/2018      # Prepare environment
mpirun -trace ${BINARY}         # Execute MPI program
traceanalyzer ${BINARY}.stf     # Analyze data
```

Example:

- ITAC: MPI benchmark rank_league 

Hands On

Preparation

- Change to folder HandsOn/Rank_league
- Use scripts `./build.itac.sh` to build rank_league benchmark
- Use `msub jobscript.itac.msub` to submit batch job
- Use `traceanalyzer rank_league.stf` to open trace file

Questions

What is shown in

- Flat Profile?
- Load Balance?
- Call Tree?

What is shown in graphical tools

- Event timeline?
- Quantitative timeline?
- Function profile?
- Message profile?

References: Benchmarks



DGEMM benchmark from Sandia National Laboratories





<http://www.nersc.gov/research-and-development/apex/apex-benchmarks/dgemm/>



Stream benchmark original version; John D. McCalpin

<https://www.cs.virginia.edu/stream/>

References: Performance Tools

-  Homepage: Application Performance Snapshot
<https://software.intel.com/sites/products/snapshots/application-snapshot/>
-  Homepage: Intel Trace Analyzer and Collector
<https://software.intel.com/en-us/intel-trace-analyzer>
-  Github-page: Likwid
<https://github.com/RRZE-HPC/likwid>
-  Homepage: Time
<https://directory.fsf.org/wiki/Time>