Karlsruhe Institute of Technology

Steinbuch Centre for Computing

Introduction to Performance Engineering on HPC

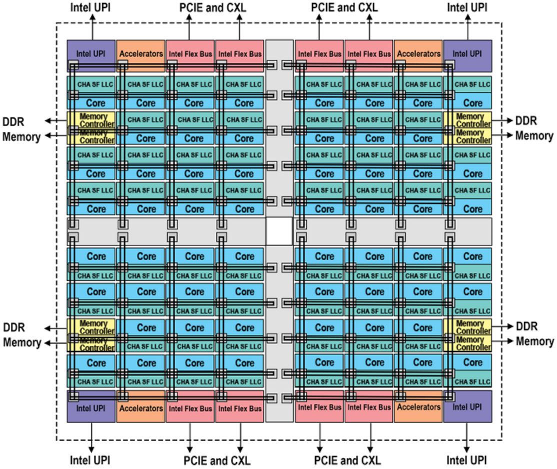
Holger Obermaier | 14. June 2023

Optimization cycle

Current hardware challenges

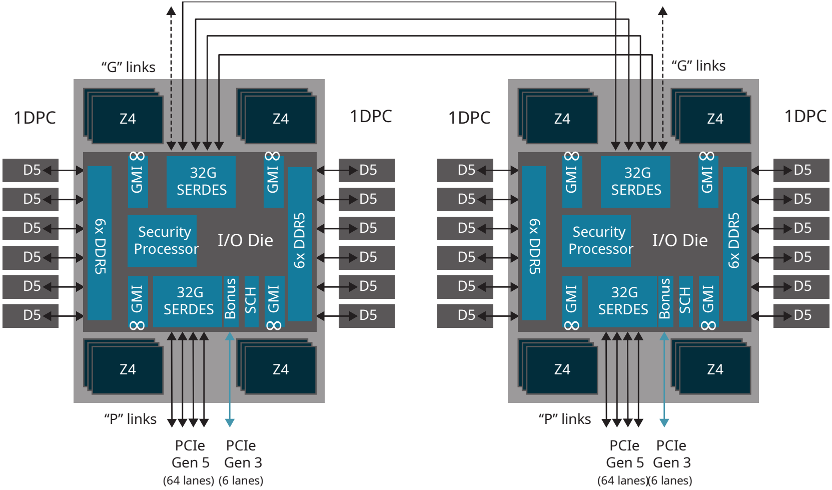
- Cooling / power restrictions
  - ⇒ CPU frequency is limited
  - ⇒ More cores
- Die size restrictions
  - ⇒ Number of logic circuits per die is limited
  - ⇒ Multiple dies per CPU
  - ⇒ Multiple communication networks between cores (on die, inter die)
- Limited number of electrical connections between CPU and board
  - ⇒ Limited number of memory channels, limited memory bandwidth
  - ⇒ Multiple levels of caches
  - ⇒ Multiple types of memory (Main memory, High bandwidth memory (HBM))

Optimization cycle (Intel Sapphire Rapids)



Picture Intel [1]

Optimization cycle (AMD Genoa)



Picture AMD [2]

Optimization cycle ...

Hardware implied software challenges

- HPC software needs hardware awareness
  - Levels of parallelization (SIMD/Vector-Instructions, Threads, MPI tasks)
  - CPU to CPU locality
  - CPU to memory locality
  - Non Uniform Memory Access (NUMA)
  - Cache sizes and hierarchy
  - Memory types

⇒ Optimizing code gets more complex

⇒ Support by performance tools is needed

Optimization cycle ...

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 ...

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* [2]

$$\begin{aligned} \text{Copy} \quad & c = a, \quad a, c \in \mathbb{R}^n \\ \text{Scale} \quad & b = \alpha c, \quad b, c \in \mathbb{R}^n, \quad \alpha \in \mathbb{R} \\ \text{Add} \quad & c = a + b, \quad a, b, c \in \mathbb{R}^n \\ \text{Triad} \quad & a = b + \alpha c, \quad a, b, c \in \mathbb{R}^n, \quad \alpha \in \mathbb{R} \end{aligned}$$

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

⇒ Memory bandwidth bound

Tool Test Cases

Benchmark *dgemm* [1]

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

- $\mathcal{O}(n^2)$ 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 that cluster *HoreKa*↗ and *bwUniCluster 2.0*↗ 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

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 Legacy)

- Usage vectorization report

```
module add compiler/intel/2022
icc ${OPT_FLAGS} \
    -qopt-report=5 \
    -qopt-report-phase=vec \
    -qopt-report-stdout \
    ${SOURCE} -o ${OUTFILE}
```

Example

Intel vectorization report: stream

Compiler Vectorization Report (Intel LLVM based)

- Usage vectorization report

```
module add compiler/intel/2022.0.2_llvm
icx ${OPT_FLAGS} \
    -qopt-report=max \
    ${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 348) mentioned twice in the Intel vectorization report?
- Why is no peel loop needed for the loop in `tuned_STREAM_Scale` (line 348)?

`/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`

Hands On

Preparation

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

Questions

- What is the difference between the two stream benchmark runs in `jobscript.time.sh`?
- 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
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 ↗
- APS Rank-to-rank communication matrix HTML report: rank_league ↗

Hands On

Preparation

- Change to folder `HandsOn/Stream`
- Use script `./build.sh` to build stream benchmark
- Use `sbatch jobscript.aps.sh` 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 ⇒ 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 `sbatch jobscript.perfctr.sh` to submit batch job

Questions

- What is the difference between the two stream benchmark runs in `jobscript.perfctr.sh`?
- 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

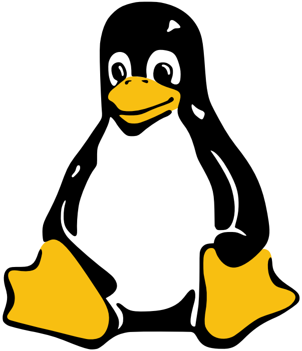
- Compare stream source code in folders `HandsOn/Stream` and `HandsOn/Stream.likwid`
- Change to folder `HandsOn/Stream.likwid`
- Use scripts `./build.gnu.sh` and `./build.intel.sh` to build stream benchmark
- Use `sbatch jobscript.gnu.sh` and `sbatch jobscript.intel.sh` to submit batch jobs

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 `sbatch jobscript.perf.sh` 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

```
# Prepare environment
source /software/all/toolkit/Intel_OneAPI/setvars.sh
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 `sbatch jobscript.itac.sh` 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

Hardware

- Technical Overview Of The 4th Gen Intel® Xeon® Scalable processor family
- 4th Gen AMD EPYC Processor Architecture

Benchmarks

- DGEMM benchmark from Sandia National Laboratories
- Stream benchmark original version; John D. McCalpin

References

Performance Tools

- Intel® VTune™ Profiler (Application Performance Snapshot),
  Get Started with Application Performance Snapshot - Linux* OS
- Intel® Trace Analyzer and Collector,
  Get Started with Intel® Trace Analyzer and Collector,
  Intel® Trace Analyzer and Collector User and Reference Guide
- LIKWID Performance Tools
  Github-page: Likwid
- GNU Time