Introduction to Performance Optimization and Tuning Tools

Steve Lantz, Cornell University

CoDaS-HEP Summer School, July 19, 2023

with thanks to Bei Wang, NVIDIA
• Give an overview of what is meant by performance optimization and tuning
• Provide basic guidance on how to understand the performance of a code using tools
• Provide a starting point for performance optimizations
Performance Tuning: What Is It? Why Do It?

• What is performance tuning?
  – The process of improving the efficiency of an application to make better use of a given hardware resource
  – A cycle of identifying bottlenecks, eliminating these where possible, and rechecking efficiency – usually continued until performance objectives are satisfied
  – Writing code informed by one’s understanding of the performance features of the given hardware (see previous presentations on “What Every Computational Physicist Should Know About Computer Architecture” and “Vector Parallelism on Multi-Core Processors”)

• Why does performance matter?
  – Energy efficiency is becoming increasingly important
  – Today’s applications only use a fraction of the machine
  – Due to complex architectures, mapping applications onto architectures is hard
The Performance Tuning Cycle

- Change only **one thing at a time**
- Consider the ease (difficulty) of implementation
- Keep **track** of all changes
- Apply regression test to ensure **correctness** after each change
- Remember: fast computing of a wrong result is completely irrelevant

- Choose a workload which is measurable, representative, static, reproducible, and quantifiable
- Record code generation, compiler version, compiler flags, input parameters, core count, affinity, etc.

1. **prepare**
2. **modify**
3. **measure**
4. **analyze**
5. **hypothesize**
What Do I Measure?

• Choose metrics which quantify the performance of your code
  – **Time** spent at different levels: whole program, functions, lines of code
  – **Hardware counters** can help you figure out the reasons for slow spots

• What are some easy ways to make time measurements?
  – Wrap your executable command in the Linux “time” command
    • Get an idea of overall run time: `time ./my_exe` (or `/bin/time ./my_exe`)
    • No way to zero in on performance bottlenecks
  – Insert calls to timers around critical loops/functions
    • `gettimeofday()`, `MPI_Wtime()`, `omp_get_wtime()`
    • Available in common libraries (system, MPI, OpenMP respectively)
    • Good for checking known hotspots in a small code base
    • Hard to maintain, require significant a priori knowledge of the code
Advantages of Performance Tools

• Performance tools (recommended)
  – Collect a lot data with varying granularity, cost and accuracy
  – Connect back to the source code (use -g compiler flag)
  – Analyze/visualize collected data using the tool
  – The learning curve is steep, but you can climb it gradually

• Tools generally work in one of two ways

  **Sampling**
  • Records system state at periodic intervals
  • Useful to get an overview
  • Low and uniform overhead
  • Ex. Profiling

  **Instrumentation**
  • Records all events
  • Provide detailed per event information
  • High overhead for request events
  • Ex. Tracing
Performance Tools Overview

• Basic OS tools
  – /bin/time
  – perf, gprof, igprof (from HEP)
  – valgrind, callgrind

• Hardware counters
  – PAPI API & tool set

• Community open source
  – HPCToolkit (Rice Univ.)
  – TAU (Univ. of Oregon)
  – Open|SpeedShop (Krell)

• Commercial products
  – Linaro Forge (DDT, MAP)

• Vendor supplied (free)
  – Intel Advisor, Intel VTune
  – Intel Trace Analyzer and Collector (MPI)
  – AMD μProf
  – CrayPat
  – NVIDIA Nsight Compute (CUDA)
  – NVIDIA pgprof (OpenACC)
  – AMD Omniprof (ROC)

No tool can do everything. Choose the right tool for the right task.
What Can I Learn From Performance Tools?

• Where am I spending my time?
  – Find the hotspots

• Is my code memory bound or compute bound?
  – Memory bound code has lots of events like these (tracked by hardware counters):
    • L1/L2/L3 cache misses
    • TLB misses
  – Compute bound code has lots of events like these:
    • Pipeline stalls not due to memory events
    • Type conversions
    • Time spent in unvectorized loops

• Is my I/O inefficient?
Typical Performance Pitfalls on a Single Node

- Scattered memory accesses that constantly bring in new cache lines
  - Storing data as an array of structs (AoS) instead of a struct of arrays (SoA)
  - Looping through arrays with a large stride

| More cache lines ⇒ data must be fetched from more distant caches, or from RAM |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| Registers       | L1              | L2              | LLC             | DRAM            |
| Speed (cycle)   | 1               | ~4              | ~10             | ~30             | ~200            |
| Size            | < KB            | ~32KB           | ~256KB          | ~35MB           | 10-100GB        |

- Mismatched types in assignments

float x=3.14; //bad: 3.14 is a double
float s=sin(x); //bad: sin() is a double precision function
long v=round(x); //bad: round() takes and returns double

float x=3.14f; //good: 3.14f is a float
float s=sinf(x); //good: sin() is a single precision function
long v=lroundf(x); //good: lroundf() takes float and returns long
Typical Performance Pitfalls: Multithreading

- Load imbalance
- False sharing: when CPUs alter different variables in the same cache line ↓
  - Data aren’t really shared, but caches must stay coherent
  - Data always travel together in “cache lines” of 64 bytes
- Insufficient parallelism
- Synchronization
  - Use private thread storage to avoid synchronization
- Non-optimal memory placement
  - Memory is actually allocated on first touch
  - Thread that touches first has fastest access

Linux Tool: *perf*

- **Perf** is a performance analyzing tool in Linux
  - *perf record*: measure and save sampling data for a single program
    - `-g`: enable call-graph (callers/callee information)
  - *perf report*: analyze the file generated by perf record, can be flat profile or graph
    - `-g`: enable call-graph (callers/callee information)
  - *perf stat*: measure total event count for a single program
    - `-e event-name-1,event-name-2`: choose from event names provided by *perf list*
  - *perf list*: list available hardware and software events for measurement

- When compiling the code, use the following flags for easier interpretation
  - `-g`: generate debug symbols needed to annotate source
  - `-fno-omit-frame-pointer`: provide stack chain/backtrace

https://perf.wiki.kernel.org/index.php/Tutorial
https://www.brendangregg.com/perf.html
**Example: Finding Hot Spots with perf**

- Compile the code: `g++ -g -fno-omit-frame-pointer -O3 -DNAIVE matmul_2D.cpp -o mm_naive.out`
- Collect profiling data: `perf record -g ./mm_naive.out 500`
- Open the result: `perf report -g`

Press “A” →

```
$ g++ -g -fno-omit-frame-pointer -O3 -DNAIVE matmul_2D.cpp -o mm_naive.out
$ perf record -g ./mm_naive.out 500
$ perf report -g
```
Example: Counting Cache Misses with *perf stat*

- **The *perf list* command lists all available CPU counters**
  - Check *man perf_event_open* to see what each event measures

- **The *perf stat* command instruments and summarizes selected CPU counters**

```
perf stat -e cpu-cycles,instructions,L1-dcache-loads,L1-dcache-load-misses ./mm_naive.out 500
```

Performance counter stats for './mm_naive.out 500':

```
5,564,503,540 cpu-cycles
10,063,662,841 instructions  # 1.81 insns per cycle
3,767,490,743 L1-dcache-loads
1,475,374,174 L1-dcache-load-misses  # 39.16% of all L1-dcache hits
```

- Make changes, see if L1 load misses improve, e.g.
Two very useful analyses in Intel Advisor will be highlighted:

• **Vectorization advisor**
  – Provide vectorization information from vectorization report
  – Identify the hotspots where your efforts pay off the most
  – Provide call graph information
  – Identify the performance and vectorization issues
  – Check memory access pattern, dependencies, more

• **Roofline**
  – How much performance is being left on the table
  – Where are the bottlenecks
  – Which can be improved
  – Which are worth improving
Workflow of Vectorization Advisor

- **Survey**: find the vectorization information for loops and provide suggestions for improvement
- **Trip Counts**: generate a **Roofline** Chart
- **Memory Access Patterns (MAP)**: see how you access the data
- **Dependencies**: determine if it is safe to force vectorization
Possible inefficient memory access patterns present

Inefficient memory access patterns may result in significant vector code execution slowdown or block automatic vectorization by the compiler. Improve performance by investigating.

Confirm inefficient memory access patterns

There is no confirmation inefficient memory access patterns are present. To fix: Run a Memory Access Patterns analysis.

Data type conversions present

There are multiple data types within loops. Utilize intrinsic vectorization support more effectively by avoiding data type conversion.

Use the smallest data type

The source loop contains data types of different widths. To fix: Use the smallest data type that gives the needed precision to use the entire vector register width.
Roofline Analysis: What Is It?
Towards Peak Flop/s: Arithmetic Intensity

• Arithmetic intensity or AI is the number of flops executed by a code divided by the bytes of memory that are required to perform the computations
  – AI is an intrinsic property of the code

• Even a simple stride-1 loop may not get the peak flop/s rate, if its AI is low
  – VPU becomes stalled waiting for loads and stores to complete
  – Delays become longer as the memory request goes further out in the hierarchy from L1 to L2 (to L3?) to RAM
  – Even if the right vectors are in L1 cache, there is limited bandwidth from L1 to registers!

• If the goal is to maximize flop/s, you’ll want to try to improve AI

• Also want threads to work on independent, cache-size chunks of data
  – Watch out for false sharing, where 2 threads fight needlessly over a cache line
Data taken on a laptop (2.6 GHz, vector width 8):

Function call overhead

Vector too small

Division – cache does not matter

L1 drop off, 32KB

L2 drop off, 256KB

L3 drop off (6MB), too soon? Output matters, too!

Effect of AI and Caches on GFLOP/s
What Does Roofline Analysis Tell You?

• Roofline analysis is a way of telling whether a piece of code is *compute bound* or *memory bound*
  – The “roofline” is a performance ceiling related to *hardware characteristics*

• The *arithmetic intensity* or AI (flop/byte) of a code tells you what part of the roof the code is under
  – AI is a *software characteristic* telling you the extent to which the code is limited by its need to load and store data from/to memory

• The roofline sets the highest flop/s rate possible for a given piece of code
  – If some of your functions fall way below that rate, you may need to investigate why
  – It’s possible to show that the AI needed for reaching *theoretical peak* flop/s (the highest flat roof) implies that 50% of operands are vector constants, i.e., they are loaded just once and never leave registers!
Intel VTune

• Covers all aspects of execution
  – Hotspots
  – Processor microarchitecture
  – Memory accesses
  – Threading
  – I/O

• Flexible
  – GUI in Linux, Windows and macOS
  – Drills down to source code, assembly
  – Easy setup, no special compiling

• Shared memory only
  – Serial or OpenMP
  – MPI, but only within a single node

**HPC Performance Characterization**

- **Elapsed Time**: 3.383s
  - SP GFLOPS: 0.000
  - DP GFLOPS: 2.673
  - x86 GFLOPS: 0.000

- **Effective CPU Utilization**: 7.31%
  - Average Effective CPU Utilization: 2.33% out of 32

- **Serial Time** (outside parallel regions): 0.002s (1.84%)
- **Parallel Region Time**: 3.229s (98.2%)
  - Estimated Ideal Time: 1.894s (56.6%)
  - OpenMP Potential Gain: 1.428s (42.1%)
  - Top OpenMP Region by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

- **Memory Bound**: 33.7% of Pipeline Slots
  - Cache Bound: 16.2% of Cachecticks
  - DRAM Bound: 0.6% of Cachecticks
  - NUMA: % of Remote Accesses: 0.0%
  - Bandwidth Utilization Histogram

- **Vectorization**: 0.0% of Packed FP Operations
  - Vector Instruction Mix:
    - SP FLUOPS: 0.0% of uops
    - DP FLUOPS: 22.9% of uops
    - Packed: 0.0% from DP FP
    - Scalar: 100.0% from DP FP
    - x87 FLUOPS: 0.0% of uops
    - Non-FP: 77.1% of uops
    - FP Arithmetic Rd Inst: 0.545
    - FP Arithmetic Wr Inst: 1.714

Top Loops/Functions with FPU Usage by CPU Time
This section provides information for the most time consuming loops/functions with floating point operations.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU Time (%)</th>
<th>% of FP Ops</th>
<th>FP Ops: Packed</th>
<th>FP Ops: Scalar</th>
<th>Vector Instruction Set</th>
<th>Loop Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Loop at line 48 in compute_transparens in parallel for ]</td>
<td>7.307s</td>
<td>28.9%</td>
<td>0.0%</td>
<td>100.0%</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

*N/A is applied to non-summable metrics.*
Hotspots Analysis

- **Elapsed Time:** 3.288s
  - **CPU Time:** 12.818s
  - **Overhead Time:** 9.569s
  - **Effective Time:** 4.249s
  - **Microarchitecture Usage:** 24.2% of Pipeline Slots
  - **CPU Rate:** 1.111
  - **Total Thread Count:** 4
  - **Passed Time:** 85

- **Top Hotsops**
  This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

- **Effective CPU Utilization Histogram**
  This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU utilization value.

- **Collection and Platform Info**
  This section provides information about the collection, including result set name and collection platform data.

**Hotspots Insights**
If you see significant hotspots in the Top Hotsops list, switch to the Function view for in-depth analysis per function. Otherwise, use the Callers/Callees view to track critical paths for these hotspots.

**Explore Additional Insights**
- **Parallels:** 7.9% (0.229 out of 32 logical CPUs)
- Use the Memory window to explore memory opportunities to increase parallelism in your application.
- **Microarchitecture Usage:** 24.2%
- Use the Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.
- **Vector Register Utilization:** 32.8%
- Use the Intel Advisor to work more on vectorization efficiency of your application.
Thread Timelines Showing “Spin and Overhead”
CPU Utilization by Threads

- **Elapsed Time**: 3.270s
- **Effective CPU Utilization**: 11.4% (3.640 out of 32 logical CPUs)

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time add to the idle CPU utilization value.

OpenMP Analysis, Collection Time: 3.270
- Serial Time (outside parallel region): 0.072s (2.2%)
- Parallel Region Time: 3.198s (97.8%)
- Estimating Ideal Time: 1.954s (56.7%)
- OpenMP Potential Gain: 1.357s (43.2%)
- Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

Total Thread Count: 4

- Inactive Wait Time with poor CPU Utilization: 0.015s (100.0% from Inactive Wait Time)
- Inactive Sync Wait Time: 0.002s
- Preemption Idle Time: 0.000s

Top Functions by Inactive Wait Time with Poor CPU Utilization:

<table>
<thead>
<tr>
<th>Function</th>
<th>Inactive Wall Time</th>
<th>Inactive Sync Wall Time</th>
<th>Inactive Wall Count</th>
<th>Preemption Wall Time</th>
<th>Preemption Wall Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>__libc_start_main+0x170b</td>
<td>0.003s</td>
<td>0.003s</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>__libc_start_main+0x16c</td>
<td>0.002s</td>
<td>0.002s</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>__libc_start_main+0x16a</td>
<td>0.001s</td>
<td>0.001s</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libcShared+0xb4fa8</td>
<td>0.001s</td>
<td>0.001s</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libcShared+0xb6f0e</td>
<td>0.001s</td>
<td>0.001s</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*NA*: not applicable to non-parsable regions.