PATTERN-DRIVEN PERFORMANCE ENGINEERING PROCESS

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 Benchmarking

Preparation
- Reliable timing (Minimum time which can be measured?)
- Document code generation (Flags, Compiler Version)
- Get exclusive System
- System state (Clock, Turbo mode, Memory, Caches)
- Consider to automate runs with a skript (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 (Baseline!)
Best Practices Benchmarking cont.

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)
Process vs. Tool

Reduce complexity!

We propose a human driven process to enable a systematic way to success!

- Executed by humans.
- Uses tools by means of data acquisition only.

Uses one of the most powerful tools available:

**Your brain!**

You are a investigator making sense of what’s going on.
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

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

Performance patterns are typical performance limiting motifs

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

Performance patterns are typical performance limiting motifs.
Performance Engineering Process: Modelling

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

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

Note: May be skipped!
Performance Engineering Process: Optimization

- **Pattern**
- **Performance Model**
- **Optimize for better resource utilization**
- **Eliminate non-expedient activity**

**Step 3 Optimization:** Improve utilization of offered resources.
Performance pattern classification

1. Maximum resource utilization
2. Hazards
3. Work related (Application or Processor)

The system offers two basic resources:

- Execution of instructions (primary)
- Transferring data (secondary)
## Patterns (I): Bottlenecks & hazards

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Performance behavior</th>
<th>Metric signature, LIKVID performance group(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bandwidth saturation</strong></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><strong>ALU saturation</strong></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><strong>Inefficient data access</strong></td>
<td>Excess data volume</td>
<td>Low BW utilization / Low cache hit ratio, frequent CL evicts or replacements (CACHE, DATA, MEM)</td>
</tr>
<tr>
<td></td>
<td>Latency-bound access</td>
<td>Relevant events are very hardware-specific, e.g., memory aliasing stalls, conflict misses, unaligned LD/ST, requeue events</td>
</tr>
<tr>
<td><strong>Micro-architectural anomalies</strong></td>
<td>Large discrepancy from simple performance model based on LD/ST and arithmetic throughput</td>
<td></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>Bad or no scaling across NUMA domains, performance improves with interleaved page placement</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><strong>Load imbalance / serial fraction</strong></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><strong>Synchronization overhead</strong></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><strong>Instruction overhead</strong></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><strong>Code composition</strong></td>
<td><strong>Expensive instructions</strong></td>
<td>Many cycles per instruction (CPI) if the problem is large-latency arithmetic</td>
</tr>
<tr>
<td></td>
<td><strong>Ineffective instructions</strong></td>
<td>Scalar instructions dominating in data-parallel loops (FLOPS_*, CPI)</td>
</tr>
</tbody>
</table>
Most frequent patterns (seen with scientific computing glasses)

Data transfer related:
- Memory bandwidth saturation
- Bad ccNUMA page placement

Parallelization
- Load imbalance
- Serial fraction

Code composition:
- Instruction overhead
- Ineffective instructions
- Expensive instructions

Overhead:
- Synchronization overhead

Excess work:
- Data volume reduction over slow data paths
- Reduction of algorithmic work
Pattern: Bandwidth Saturation

1. Perform scaling run inside ccNUMA domain
2. Measure memory bandwidth with HPM
3. Compare to micro benchmark with similar data access pattern

Measured bandwidth:
45964 MB/s
Microbenchmark:
47022 MB/s
Factors with bandwidth utilization:

- Prefetching working effectively
  → Pattern: Inefficient data access

- Consumed bandwidth is equal to transferred bandwidth
  → Pattern: Excess data volume

- Data volume reduction (Data reuse, Non-Temporal stores)
Pattern: Load imbalance

1. Check HPM instruction count distribution across cores
   - Instructions retired / CPI may not be a good indication of useful workload – at least for numerical / FP intensive codes….
   - Floating Point Operations Executed is often a better indicator

```c
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, I
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```
Example for a load balanced code

```c
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, N
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```

```
env OMP_NUM_THREADS=6 likwid-perfctr -C S0:0-5 -g FLOPS_DP ./a.out
```

<table>
<thead>
<tr>
<th>Event</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
<th>core 4</th>
<th>core 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR RETIRED ANY</td>
<td>1.83124e+10</td>
<td>1.74784e+10</td>
<td>1.68453e+10</td>
<td>1.66794e+10</td>
<td>1.76685e+10</td>
<td>1.91736e+10</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>2.24797e+10</td>
<td>2.23789e+10</td>
<td>2.23802e+10</td>
<td>2.23808e+10</td>
<td>2.23799e+10</td>
<td>2.23805e+10</td>
</tr>
<tr>
<td>CPU CLK UNHALTED REF</td>
<td>2.04416e+10</td>
<td>2.03445e+10</td>
<td>2.03465e+10</td>
<td>2.03462e+10</td>
<td>2.03453e+10</td>
<td>2.03459e+10</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_PACKED</td>
<td>3.45348e+09</td>
<td>3.43035e+09</td>
<td>3.37573e+09</td>
<td>3.39272e+09</td>
<td>3.26132e+09</td>
<td>3.2377e+09</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_SCALAR</td>
<td>2.93108e+07</td>
<td>3.06063e+07</td>
<td>2.9704e+07</td>
<td>2.96507e+07</td>
<td>2.41141e+07</td>
<td>2.37397e+07</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION</td>
<td></td>
<td></td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION</td>
<td>3.48279e+09</td>
<td>3.46096e+09</td>
<td>3.40543e+09</td>
<td>3.42237e+09</td>
<td>3.28543e+09</td>
<td>3.26144e+09</td>
</tr>
</tbody>
</table>

Higher CPI but better performance
Pattern: Instruction Overhead

1. Perform a HPM instruction decomposition analysis
2. Measure resource utilization
3. Static code analysis

<table>
<thead>
<tr>
<th>Instruction decomposition</th>
<th>Inlining failed</th>
<th>Inefficient data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic FP</td>
<td>12%</td>
<td>21%</td>
</tr>
<tr>
<td>Load/Store</td>
<td>30%</td>
<td>50%</td>
</tr>
<tr>
<td>Branch</td>
<td>24%</td>
<td>10%</td>
</tr>
<tr>
<td>Other</td>
<td>34%</td>
<td>19%</td>
</tr>
</tbody>
</table>

C++ codes which suffer from overhead (inlining problems, complex abstractions) need a lot more overall instructions related to the arithmetic instructions

- Often (but not always) “good” (i.e., low) CPI
- Low-ish bandwidth
- Low # of floating-point instructions vs. other instructions
Pattern: Inefficient Instructions

1. HPM measurement: Relation packed vs. scalar instructions
2. Static assembly code analysis: Search for scalar loads

There is usually no counter for packed vs scalar (SIMD) loads and stores.
Also the compiler usually does not distinguish!

Only solution: Inspect code at assembly level.
Pattern: Synchronization overhead

1. Performance is decreasing with growing core counts
2. Performance is sensitive to topology
3. Static code analysis: Estimate work vs. barrier cost.

![Graph showing performance vs. number of threads]

- sync overhead grows with # of threads
- bandwidth scalability across memory interfaces
Computer Tomography reconstruction

General motivation: Reconstruction of 3D data from 2D X-ray projection images.

Here: X-ray projections acquired during an intervention.

Reconstruction should be as fast as possible:
- Interventional
- Time resolved (4D) reconstruction

Method: 3D cone beam reconstruction of high resolution C-arm Computer Tomography dataset
Why is RabbitCT interesting?

• Strong scaling problem
• Volume with $512^3$ voxels ($1024^3$ also available)
• 496 projection images (1248x960px) (2.4GB)
• 13 additions, 5 subtractions, 17 multiplications, 3 divides (can be reduced to 1 reciprocal), single precision
• Data volume on volume 496GB, **streaming pattern**
• **Non-deterministic** data access **pattern** on projection images
• Non-trivial arithmetic (profits from FMA)
• On current architectures **instruction throughput limited**

• Demanding target for SIMD vectorization
• Popular optimization target for GPUs
Example rabbitCT

Result of effort:
5-6 x improvement against initial parallel C code implementation

>50% of peak performance (SSE)
Optimization without knowledge about bottleneck
CASE STUDY: HPCCG

Performance analysis on:
- Intel IvyBridge-EP@2.2GHz
- Intel Xeon Phi@1.05GHz
for(int k=1; k<max_iter && normr > tolerance; k++)
{
    oldrtrans = rtrans;
    ddot (nrow, r, r, &rtrans, t4);
    double beta = rtrans/oldrtrans;
    waxpby (nrow, 1.0, r, beta, p, p);
    normr = sqrt(rtrans);
    HPC_sparsemv(A, p, Ap);
    double alpha = 0.0;
    ddot(nrow, p, Ap, &alpha, t4);
    alpha = rtrans/alpha;
    waxpby(nrow, 1.0, r, -alpha, Ap, r);
    waxpby(nrow, 1.0, x, alpha, p, x);
    niters = k;
}
Components of HPCCG 1

ddot:
#pragma omp for reduction (+:result)
for (int i=0; i<n; i++) {
    result += x[i] * y[i];
}

waxpby:
#pragma omp for
for (int i=0; i<n; i++) {
    w[i] = alpha * x[i] + beta * y[i];
}
Components of HPCCG 2

```c
#pragma omp for
for (int i=0; i< nrow; i++) {
    double sum = 0.0;
    double* cur_vals = vals_in_row[i];
    int* cur_inds = inds_in_row[i];
    int cur_nnz = nnz_in_row[i];

    for (int j=0; j< cur_nnz; j++) {
        sum += cur_vals[j]*x[cur_inds[j]];
    }
    y[i] = sum;
}
```
First Step: Runtime Profile \((300^3)\)

**Intel IvyBridge-EP (2.2GHz, 10 cores/chip)**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Serial</th>
<th>Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddot</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>waxby</td>
<td>12%</td>
<td>16%</td>
</tr>
<tr>
<td>spmv</td>
<td>83%</td>
<td>79%</td>
</tr>
</tbody>
</table>

**Intel Xeon Phi (1.05GHz, 60 cores/chip)**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddot</td>
<td>3%</td>
</tr>
<tr>
<td>waxby</td>
<td>8%</td>
</tr>
<tr>
<td>spmv</td>
<td>89%</td>
</tr>
</tbody>
</table>
Scaling behavior inside socket (IvyBridge-EP)

HPM measurement with LIKWID instrumentation on socket level

<table>
<thead>
<tr>
<th>Routine</th>
<th>Time [s]</th>
<th>Memory Bandwidth [MB/s]</th>
<th>Data Volume [GB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>waxby 1</td>
<td>2.33</td>
<td>40464</td>
<td>93</td>
</tr>
<tr>
<td>waxby 2</td>
<td>2.37</td>
<td>39919</td>
<td>94</td>
</tr>
<tr>
<td>waxby 3</td>
<td>2.4</td>
<td>40545</td>
<td>96</td>
</tr>
<tr>
<td>ddot 1</td>
<td>0.72</td>
<td>46886</td>
<td>34</td>
</tr>
<tr>
<td>ddot 2</td>
<td>1.4</td>
<td>46444</td>
<td>64</td>
</tr>
<tr>
<td>spmv</td>
<td>33.84</td>
<td>45964</td>
<td>1555</td>
</tr>
</tbody>
</table>

Pattern: Bandwidth saturation
Scaling to full node \((180^3)\)

### Performance [GFlops/s]

<table>
<thead>
<tr>
<th>Routine</th>
<th>Socket</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddot</td>
<td>6726</td>
<td>14547</td>
</tr>
<tr>
<td>waxby</td>
<td>3642</td>
<td>6123</td>
</tr>
<tr>
<td>spmv</td>
<td>6374</td>
<td>6320</td>
</tr>
<tr>
<td>Total</td>
<td>5973</td>
<td>6531</td>
</tr>
</tbody>
</table>

### Memory Bandwidth measured [GB/s]

<table>
<thead>
<tr>
<th>Routine</th>
<th>Socket 1</th>
<th>Socket 2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddot</td>
<td>44020</td>
<td>47342</td>
<td>91362</td>
</tr>
<tr>
<td>waxby</td>
<td>39795</td>
<td>28424</td>
<td>68219</td>
</tr>
<tr>
<td>spmv</td>
<td>43109</td>
<td>2863</td>
<td>45972</td>
</tr>
</tbody>
</table>

Pattern: Bad ccNUMA page placement
Optimization: Apply correct data placement

Matrix data was not placed. **Solution:** Add first touch initialization.

```c
#pragma omp parallel for
    for (int i=0; i< local_nrow; i++){
        for (int j=0; j< 27; j++) {
            curvalptr[i*27 + j] = 0.0;
            curindptr[i*27 + j] = 0;
        }
    }
```

Node performance: spmv 11692, total 10912

<table>
<thead>
<tr>
<th>Routine</th>
<th>Socket 1</th>
<th>Socket 2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddot</td>
<td>46406</td>
<td>48193</td>
<td>94599</td>
</tr>
<tr>
<td>waxby</td>
<td>37113</td>
<td>24904</td>
<td>62017</td>
</tr>
<tr>
<td>spmv</td>
<td>45822</td>
<td>40935</td>
<td>86757</td>
</tr>
</tbody>
</table>
Scaling behavior Intel Xeon Phi

Code is instruction throughput limited
Pattern: Expensive Instructions

134804 MB/s
131803 MB/s
BJDS (Blocked JDS) – data format

Format creation
1. Shift nonzeros in each row to the left
2. Combine `chunkHeight` (multiple of vector length, here: 8) rows to one chunk
3. Pad all rows in chunk to the same length
4. Store matrix chunk by chunk and jagged-diagonal-wise within chunk

Data arrays
- `double val[]`
- `unsigned int col[]`
- `unsigned int chunkStart[]`
Optimized spmv data structure on Xeon Phi

Pattern: Bandwidth saturation