Hierarchical Roofline Analysis on CPUs

Charlene Yang
Lawrence Berkeley National Laboratory
ECP 2020, Houston
Outline

- Hierarchical Roofline on Intel CPUs
  - L1, L2, L3, HBM, DRAM

- Methodology for Roofline Data Collection
  - Machine characterization: peak bandwidth and peak GFLOP/s
  - Empirical Roofline Toolkit (ERT)
  - Application characterization: FLOPs, bytes, runtime
    - LIKWID, SDE, VTune

- A Stencil Example

This methodology can be extended to other CPUs, and other instruction types!
CPU Architecture: HSW

- **Goal**: Hierarchical Roofline
- **Machine Characterization**
  - compute/bandwidth peaks
- **Application Characterization**
  - Performance Throughput
    - FLOPs / runtime
  - Arithmetic Intensity
    - $\text{AI}_{\text{DRAM}} = \frac{\text{FLOPS}}{\text{Bytes}_{\text{DRAM}}}$
    - $\text{AI}_{\text{MCDRAM}} = \frac{\text{FLOPS}}{\text{Bytes}_{\text{MCDRAM}}}$
    - $\text{AI}_{\text{L2}} = \frac{\text{FLOPS}}{\text{Bytes}_{\text{L2}}}$
    - $\text{AI}_{\text{L1}} = \frac{\text{FLOPS}}{\text{Bytes}_{\text{L1}}}$

Courtesy of Zakhar Matveev
Machine Characterization

- “Theoretical Performance” numbers can be highly optimistic…
  - Pin BW vs. sustained bandwidth
  - TurboMode / Underclock for AVX
  - compiler failings on high-AI loops.

- LBL developed the Empirical Roofline Toolkit (ERT)…
  - Characterize CPU/GPU systems
  - Peak Flop rates
  - Bandwidths for each level of memory

https://bitbucket.org/berkeleylab/cs-roofline-toolkit/
https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/
Application Characterization

- How to get runtime, FLOPs, Bytes ....
  - manual counting
  - performance counters
  - binary instrumentation

- Tools we can use...
  - LIKWID: vops, low overhead, no breakdown info
  - SDE + VTune: more accurate, high overhead, manual scripting required
  - Advisor: automated, high overhead, information rich
  - ...
## How Do We Count Flop’s?

<table>
<thead>
<tr>
<th>Manual Counting</th>
<th>Perf. Counters</th>
<th>Binary Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Go thru each loop nest and count the number of FP operations</td>
<td>- Read counter before/after</td>
<td>- Automated inspection of assembly at run time</td>
</tr>
<tr>
<td>✓ Works best for deterministic loop bounds</td>
<td>✓ More Accurate</td>
<td>✓ Most Accurate</td>
</tr>
<tr>
<td>✓ or parameterize by the number of iterations (recorded at run time)</td>
<td>✓ Low overhead (&lt;%) == can run full MPI applications</td>
<td>✓ FMA-, VL-, and mask-aware</td>
</tr>
<tr>
<td>✘ Not scalable</td>
<td>✓ Can detect load imbalance</td>
<td>✓ Can count instructions by class/type</td>
</tr>
<tr>
<td></td>
<td>✘ Requires privileged access</td>
<td>✓ Can detect load imbalance</td>
</tr>
<tr>
<td></td>
<td>✘ Requires manual instrumentation (+overhead) or full-app characterization</td>
<td>✓ Can include effects from non-FP instructions</td>
</tr>
<tr>
<td></td>
<td>✘ Broken counters = garbage</td>
<td>✓ Automated application to multiple loop nests</td>
</tr>
<tr>
<td></td>
<td>✘ May not differentiate FMADD from FADD</td>
<td>✘ &gt;10x overhead (short runs / reduced concurrency)</td>
</tr>
<tr>
<td></td>
<td>✘ No insight into special pipelines</td>
<td></td>
</tr>
</tbody>
</table>
# How Do We Measure Data Movement?

## Manual Counting
- Go thru each loop nest and estimate how many bytes will be moved
- Use a mental model of caches
  - Works best for simple loops that stream from DRAM (stencils, FFTs, spare, ...)
  - N/A for complex caches
  - Not scalable

## Perf. Counters
- Read counter before/after
  - Applies to full hierarchy (L2, DRAM,)
  - Much more Accurate
  - Low overhead (<%) == can run full MPI applications
  - Can detect load imbalance
  - Requires privileged access
  - Requires manual instrumentation (+overhead) or full-app characterization

## Cache Simulation
- Build a full cache simulator driven by memory addresses
  - Applies to full hierarchy and multicore
  - Can detect load imbalance
  - Automated application to multiple loop nests
  - Ignores prefetchers
  - >10x overhead (short runs / reduced concurrency)
Roofline with LIKWID
LIKWID

- LIKWID provides easy to use wrappers for measuring performance counters...
  - **Works on NERSC production systems**
  - Works on NERSC production systems
  - Distills counters into user-friendly metrics (e.g. MCDRAM Bandwidth)
  - Minimal overhead (<1%)
  - Scalable in distributed memory (MPI-friendly)
  - Fast, high-level characterization
  - No timing breakdowns
  - Suffers from Garbage-in/Garbage Out
    - (i.e. hardware counter must be sufficient and correct)

https://github.com/RRZE-HPC/likwid
<table>
<thead>
<tr>
<th>likwid-topology</th>
<th>node topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>likwid-pin</td>
<td>process/thread affinity</td>
</tr>
<tr>
<td>likwid-memsweeper</td>
<td>cleanup memory &amp; LLC</td>
</tr>
<tr>
<td>likwid-powermeter</td>
<td>power measurements</td>
</tr>
<tr>
<td>likwid-setFrequencies</td>
<td>CPU/uncore frequency manipulation</td>
</tr>
<tr>
<td>likwid-perfctr</td>
<td>hardware counter measurements</td>
</tr>
<tr>
<td>likwid-mpirun</td>
<td>hardware counter + MPI</td>
</tr>
<tr>
<td>likwid-bench</td>
<td>micro-benchmarking</td>
</tr>
<tr>
<td>likwid-agent</td>
<td>system monitoring</td>
</tr>
<tr>
<td>likwid-genTopoCfg</td>
<td>generate and store topology file</td>
</tr>
</tbody>
</table>
LIKWID Marker API

• By default, profiles whole program
• But Marker API allows regional profiling as well

```c
#include <likwid.h>

......
LIKWID_MARKER_INIT;
#pragma omp parallel {
    LIKWID_MARKER_THREADINIT;
}
#pragma omp parallel {
    LIKWID_MARKER_START("foo");
    #pragma omp for
    for(i = 0; i < N; i++) {
        data[i] = omp_get_thread_num();
    }
    LIKWID_MARKER_STOP("foo");
}
LIKWID_MARKER_CLOSE;
```

focus on specific code regions
<table>
<thead>
<tr>
<th>Group name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBM_OFFCORE</td>
<td>Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)</td>
</tr>
<tr>
<td>TLB_INSTR</td>
<td>L1 Instruction TLB miss rate/ratio</td>
</tr>
<tr>
<td>FLOPS_SP</td>
<td>Single Precision MFLOP/s</td>
</tr>
<tr>
<td>BRANCH</td>
<td>Branch prediction miss rate/ratio</td>
</tr>
<tr>
<td>L2CACHE</td>
<td>L2 cache miss rate/ratio</td>
</tr>
<tr>
<td>ENERGY</td>
<td>Power and Energy consumption</td>
</tr>
<tr>
<td>FRONTEND_STALLS</td>
<td>Frontend stalls</td>
</tr>
<tr>
<td>ICACHE</td>
<td>Instruction cache miss rate/ratio</td>
</tr>
<tr>
<td>TLB_DATA</td>
<td>L2 data TLB miss rate/ratio</td>
</tr>
<tr>
<td>MEM</td>
<td>Memory bandwidth in MBytes/s</td>
</tr>
<tr>
<td>DATA</td>
<td>Load to store ratio</td>
</tr>
<tr>
<td>L2</td>
<td>L2 cache bandwidth in MBytes/s</td>
</tr>
<tr>
<td>FLOPS_DP</td>
<td>Double Precision MFLOP/s</td>
</tr>
<tr>
<td>CLOCK</td>
<td>Power and Energy consumption</td>
</tr>
<tr>
<td>HBM_CACHE</td>
<td>Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)</td>
</tr>
<tr>
<td>HBM</td>
<td>Memory bandwidth in MBytes/s for High Bandwidth Memory (HBM)</td>
</tr>
<tr>
<td>UOPS_STALLS</td>
<td>UOP retirement stalls</td>
</tr>
</tbody>
</table>
Example GPP: GFLOP/s

- **GPP kernel on KNL:** 171.960 GFLOPS/sec
  - UOPS_RETIRED_PACKED_SIMD
  - UOPS_RETIRED_SCALAR_SIMD

- **likwid-perfctr -C 0-63 -g FLOPS_DP ./gpp.knl.ex 512 2 32768 20**
  - 8*UOPS_RETIRED_PACKED_SIMD+UOPS_RETIRED_SCALAR_SIMD

```
<table>
<thead>
<tr>
<th>Metric</th>
<th>Sum</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s] STAT</td>
<td>940.8064</td>
<td>14.7001</td>
<td>14.7001</td>
<td>14.7001</td>
</tr>
<tr>
<td>Clock [MHz] STAT</td>
<td>96000.0155</td>
<td>1499.9955</td>
<td>1500.0007</td>
<td>1500.0002</td>
</tr>
<tr>
<td>CPI STAT</td>
<td>86.0772</td>
<td>1.3396</td>
<td>1.5850</td>
<td>1.3450</td>
</tr>
<tr>
<td>DP MFLOP/s (SSE assumed) STAT</td>
<td>44456.2105</td>
<td>688.9334</td>
<td>729.9324</td>
<td>694.6283</td>
</tr>
<tr>
<td>DP MFLOP/s (AVX assumed) STAT</td>
<td>86957.6422</td>
<td>1347.4354</td>
<td>1429.2337</td>
<td>1358.7132</td>
</tr>
<tr>
<td>DP MFLOP/s (AVX512 assumed) STAT</td>
<td>171960.5065</td>
<td>2664.4393</td>
<td>2827.8362</td>
<td>2686.8829</td>
</tr>
<tr>
<td>Packed MUOPS/s STAT</td>
<td>21250.7182</td>
<td>329.2510</td>
<td>349.6506</td>
<td>332.0424</td>
</tr>
<tr>
<td>Scalar MUOPS/s STAT</td>
<td>1954.7786</td>
<td>30.4313</td>
<td>30.6312</td>
<td>30.5434</td>
</tr>
</tbody>
</table>
```
Example GPP: MCDRAM + DDR GB/s

- kernel on KNL: DDR 2.59GB/s + MCDRAM 63.71GB/s
  - MC_CAS_READS/ MC_CAS_WRITES
  - EDC_RPQ_INSERTS/ EDC_WPQ_INSERTS
  - EDC_MISS_CLEAN/ EDC_MISS_DIRTY

likwid-perfctr -C 0-63 -g HBM_CACHE ./gpp.knl.ex 512 2 32768 20
Example GPP: L2 GB/s

- kernel on KNL: L2 96.80GB/s
  - L2_REQUESTSREFERENCE
  - OFFCORE_RESPONSE_0_OPTIONS
- likwid-perfctr -C 0-63 -g L2 ./gpp.knl.ex 512 2 32768 20

```
<table>
<thead>
<tr>
<th>Metric</th>
<th>Sum</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s] STAT</td>
<td>895.5200</td>
<td>13.9925</td>
<td>13.9925</td>
<td>13.9925</td>
</tr>
<tr>
<td>Clock [MHz] STAT</td>
<td>95999.4279</td>
<td>1499.9861</td>
<td>1499.9914</td>
<td>1499.9911</td>
</tr>
<tr>
<td>CPI STAT</td>
<td>83.8844</td>
<td>1.3055</td>
<td>1.5567</td>
<td>1.3107</td>
</tr>
<tr>
<td>L2 non-RFO bandwidth [MBytes/s] STAT</td>
<td>96803.9243</td>
<td>1498.7686</td>
<td>1904.3169</td>
<td>1512.5613</td>
</tr>
<tr>
<td>L2 RFO bandwidth [MBytes/s] STAT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L2 RFO data volume [GByte] STAT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L2 bandwidth [MBytes/s] STAT</td>
<td>96803.9243</td>
<td>1498.7686</td>
<td>1904.3169</td>
<td>1512.5613</td>
</tr>
<tr>
<td>L2 data volume [GByte] STAT</td>
<td>1.354528e+06</td>
<td>20971.5004</td>
<td>26646.1299</td>
<td>21164.4950</td>
</tr>
</tbody>
</table>
```
Example GPP: L1 GB/s

- **kernel on KNL:** L1 170.77GB/s
  - MEM_UOPS RETIRED_ALL_LOADS
  - MEM_UOPS RETIRED_ALL_STORES

- **likwid-perfctr -C 0-63 -g DATA ./gpp.knl.ex 512 2 32768 20**
  - (MEM_UOPS RETIRED_ALL_LOADS + MEM_UOPS RETIRED_ALL_STORES)*64/runtime
  - -g DATA is for load-to-store ratio, but can be used to estimate L1 bandwidth (assume all loads are vector loads)

<table>
<thead>
<tr>
<th>AI (DRAM)</th>
<th>66.39</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI (MCDRAM)</td>
<td>2.70</td>
</tr>
<tr>
<td>AI (L2)</td>
<td>1.78</td>
</tr>
<tr>
<td>AI (L1)</td>
<td>1.01</td>
</tr>
<tr>
<td>Performance</td>
<td>171.960 GFLOPS/s</td>
</tr>
</tbody>
</table>

**Attainable Flop/s**
- L1: 2.7TFLOP/s
- L2: 1.78TFLOP/s
- MCDRAM: 368.5GB/s
- DDR: 77.0GB/s

**Arithmetic Intensity**
- AI (DRAM): 66.39
- AI (MCDRAM): 2.70
- AI (L2): 1.78
- AI (L1): 1.01
- Performance: 171.960 GFLOPS/s
Intel Software Development Emulator (SDE)

- Dynamic instruction tracing
  - Accounts for actual loop lengths and branches
  - Counts instruction types, lengths, etc…
  - Can mark individual regions
  - Support for MPI+OpenMP
  - Can be used to calculate FLOPs (VL-, FMA-, and precision-aware)
  - Post processing can be expensive.
  - No insights into cache behavior or DRAM data movement
  - X86 only

When the job completes, you’ll have a series of files prefixed with “sde_”.

Parse the output to summarize the results...

```
$ ./parse-sde.sh sde_2p16t*
```

Use the “Total FLOPs” line as the numerator in all AI’s and performance

Use the “Total Bytes” line as the denominator in the L1 AI

Can infer vectorization rates and precision
LIKWID vs. SDE

- Recall, LIKWID counts vector uops while SDE counts instructions.
- Why does this matter?
  - VL-aware: KNL has scalar but treats 128b, 256b, and 512b as 512b.
  - Precision-aware: User has to know which precision they use.
  - Mask-aware: KNL counters ignore masks.
  - FMA-aware: LIKWID assumes 1 flop per element.
  - KNL counts vector integer, stores, NT stores, and gathers as vector uops (and thus as potential flop/s).

LIKWID’s and SDE’s counts of #FP ops and Gflop/s can be different (very different for linear algebra).
LIKWID vs. SDE/VTune

- **SDE FLOPS:**
  - `sde64 -knl -d -iform 1 -omix my_mix.out -global_region -- ./gpp.knl.ex 512 2 32768 20`
  - `./parse-sde.sh my_mix.out`
  - `-->Total FLOPs = 2775769815463`

- **VTune Bytes:**
  - `amplxe-cl -collect memory-access -finalization-mode=deferred -r my_vtune/ -- ./gpp.knl.ex 512 2 32768 20`
  - `amplxe-cl -report summary -r my_vtune/ > my_vtune.summary`
  - `./parse-vtune.sh my_vtune.summary`
  - `DDR -->Total Bytes = 35983553088`
  - `HBM -->Total Bytes = 963486016448`

Roofline with Advisor
The Roofline Feature in Intel® Advisor

- Automate data collection, one dot per kernel
- Hierarchical Roofline for multiple caches
- Automatically benchmarks target system
- Fully integrated with other Advisor features

A single button or CLI command runs the Survey and FLOPS analyses to generate the Roofline chart.

 Courtesy of Zakhar Matveev
## Intel Advisor: 2-pass Approach

### Roofline:

<table>
<thead>
<tr>
<th>Axis X:</th>
<th>AI = #FLOP / #Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis Y:</td>
<td>FLOP/S = #FLOP (mask aware) / #Seconds</td>
</tr>
</tbody>
</table>

### Overhead

<table>
<thead>
<tr>
<th>Step 1: Survey (-collect survey)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide #Seconds</td>
</tr>
<tr>
<td>Root access not needed</td>
</tr>
<tr>
<td>User mode sampling, non-intrusive.</td>
</tr>
</tbody>
</table>

1x

<table>
<thead>
<tr>
<th>Step 2: FLOPS (-collect tripcounts –flops)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide #FLOP, #Bytes, AVX-512 Mask</td>
</tr>
<tr>
<td>Root access not needed</td>
</tr>
<tr>
<td>Precise, instrumentation based, count number of instructions</td>
</tr>
</tbody>
</table>

5-10x
Intel Advisor: Command Lines for Roofline

$ source advixe-vars.sh

1st method. Not compatible with MPI applications:
$ advixe-cl -collect roofline --project-dir ./dir -- ./app

2nd method (old, more flexible):
$ advixe-cl -collect survey --project-dir ./dir -- ./app
$ advixe-cl -collect tripcounts -flop --project-dir ./dir -- ./app

(optional) copy data to your UI desktop system
$ advixe-gui ./dir

IRM How-to:
Intel Advisor: A Stencil Example Iso3DFD

For (int iz=0; iz<n3; iz++)
For (int iy=0; iy<n2; iy++)
For (int ix=0; ix<n1; ix++) {
    int offset = iz*dimn1n2 + iy*n1 + ix;
    float value = 0.0;
    value += ptr_prev[offset]*coeff[0];
    for(int ir=1; ir<= 8 ; ir++) {
        value += coeff[ir] * (ptr_prev[offset + ir] + ptr_prev[offset - ir]);
        value += coeff[ir] * (ptr_prev[offset + ir*n1] + ptr_prev[offset - ir*n1]);
        value += coeff[ir] * (ptr_prev[offset + ir*dimn1n2] + ptr_prev[offset - ir*dimn1n2]);
    }
    ptr_next[offset] = 2.0f* ptr_prev[offset] - ptr_next[offset] + value*ptr_vel[offset];
}
Intel Advisor: A Stencil Example Iso3DFD

Progressive levels of optimization

- Dev00: unoptimized implementation of iso3DFD
- Dev01: adding OpenMP threading
- Dev02: reverse loops improving memory access pattern
- Dev03: vectorization, improve compute throughput and L1 AI
- Dev04: implement cache blocking, improving DRAM AI
v00 – where am I?

- Main hotspot is loop at iso-3dfd_parallel.cc:43
  - Performance is far from machine peak
  - Problem:
    - Serial – 1 thread (Summary, Roofline)
    - Scalar

![Program metrics](image)
v01 – introduce OpenMP threading

Top max GFLOPS limit increases with number of threads

Add v00 to comparison

The loop moves up after threading
Enable Integrated Roofline Model

1. Enable showing memory level relationships
2. Double click to see all memory levels traffic

Zoom by selection, taking the dot and upper roofs

Do you see anything suspicious?

Enable Integrated Roofline Model
v01 – Memory Access Patterns

<table>
<thead>
<tr>
<th>ID</th>
<th>Stride</th>
<th>Type</th>
<th>Nested Function</th>
<th>Variable references</th>
<th>Memory object allocation site</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>65536</td>
<td>Constant stride</td>
<td>block 0x7fd89ffe010 allocated at iso-3dfd_main.cc:184, block 0x7</td>
<td>Strided access</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>65536</td>
<td>Constant stride</td>
<td>block 0x7fd89ffe010 allocated at iso-3dfd_main.cc:184, block 0x7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
47  for(int ir=1; ir<HALF_LENGTH; ir++) {
48  value += ptr_prev[offset]*coeff[0];
49  for(int ir=1; ir<HALF_LENGTH; ir++) {
50  value += coeff[ir] * (ptr_prev[offset + ir*nl] + ptr_prev[offset - ir*nl]); // horizontal
51  value += coeff[ir] * (ptr_prev[offset + ir*dimin2] + ptr_prev[offset - ir*dimin2]); // vertical
```

- **Strided access**
- **Memory object allocation site**
v02 – reverse loops

L1, L2, LLC and DRAM are in order

All unit strides now
v02 – find reason for no vectorization

Outer loop was not auto-vectorized

Cause: The compiler vectorizer determined outer loop vectorization is not possible using auto-vectorization.

C++ Example:

```c++
void foo(float **a, float **b, int N) {
    int i, j;
    #pragma ivdep
    for (i = 0; i < N; i++) {
        float *ap = a[i];
        float *bp = b[i];
        for (j = 0; j < N; j++) {
            ap[j] = bp[j];
        }
    }
}
```
Compare all memory levels with v02

Enable all memory levels

L1 AI is higher after vectorization, due to less impact from **pointer arithmetic**

v03

v02
Integrated Roofline: DRAM is the bottleneck

CARM Roofline Guidance: either DRAM or LLC is the bottleneck.
v04 – implement cache blocking

V04: higher memory locality

V03: shorter difference in AI => less locality

DRAM is still bottleneck, but now the limit is higher

DRAM memory impact decreases after cache blocking