PERFORMANCE ENGINEERING

Lecture 2: Roofline Models and beyond

April 11th, 2022

Ana Lucia Varbanescu
a.l.varbanescu@uva.nl

What if …

- T=1s, Machine = 8-core 4-way SIMD CPU,128GFLOPs
- GLOPS? BW? Utilization?

```
struct complex {float re; float im;};
int sum_array_3d(complex a[N][N][N])
{
    int i, j, k,
    float partial[N], sum = 0;
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++) {
          if (a[k][i][j].re > 0)
              partial[i] += a[k][i][j].re
          sum += a[k][i][j].re;
         }
    return sum;
}
```

Solution

- Compute operations:
  - ?
- Memory operations:
  - ?

In general …

- Obtaining instruction mix from source code is difficult
- High accuracy: use profiling information & tools

- Potential tools of interest
  - Perf
  - nersc-roofline
  - Intel SDE
  - Intel Advisor
  - Roofline toolkit

To Do Today

- ~~Last lecture's quiz~~
- Brief recap
- Roofline model in more detail
- Overview of application performance modeling

Recap [1]

- Computing systems hardware
  - Cores
    - Multi-core vs. GPUs
  - Memory systems
    - The memory hierarchy
  - Performance
    - Compute peak [FLOPS]
    - Bandwidth peak [GB/s]
    - Platform/machine balance = peak_FLOPS/peak_BW [FLOPS/byte]

Theoretical peak performance

Throughput [GFLOPs] = chips * cores * vectorWidth * FLOPs/cycle * clockFrequency

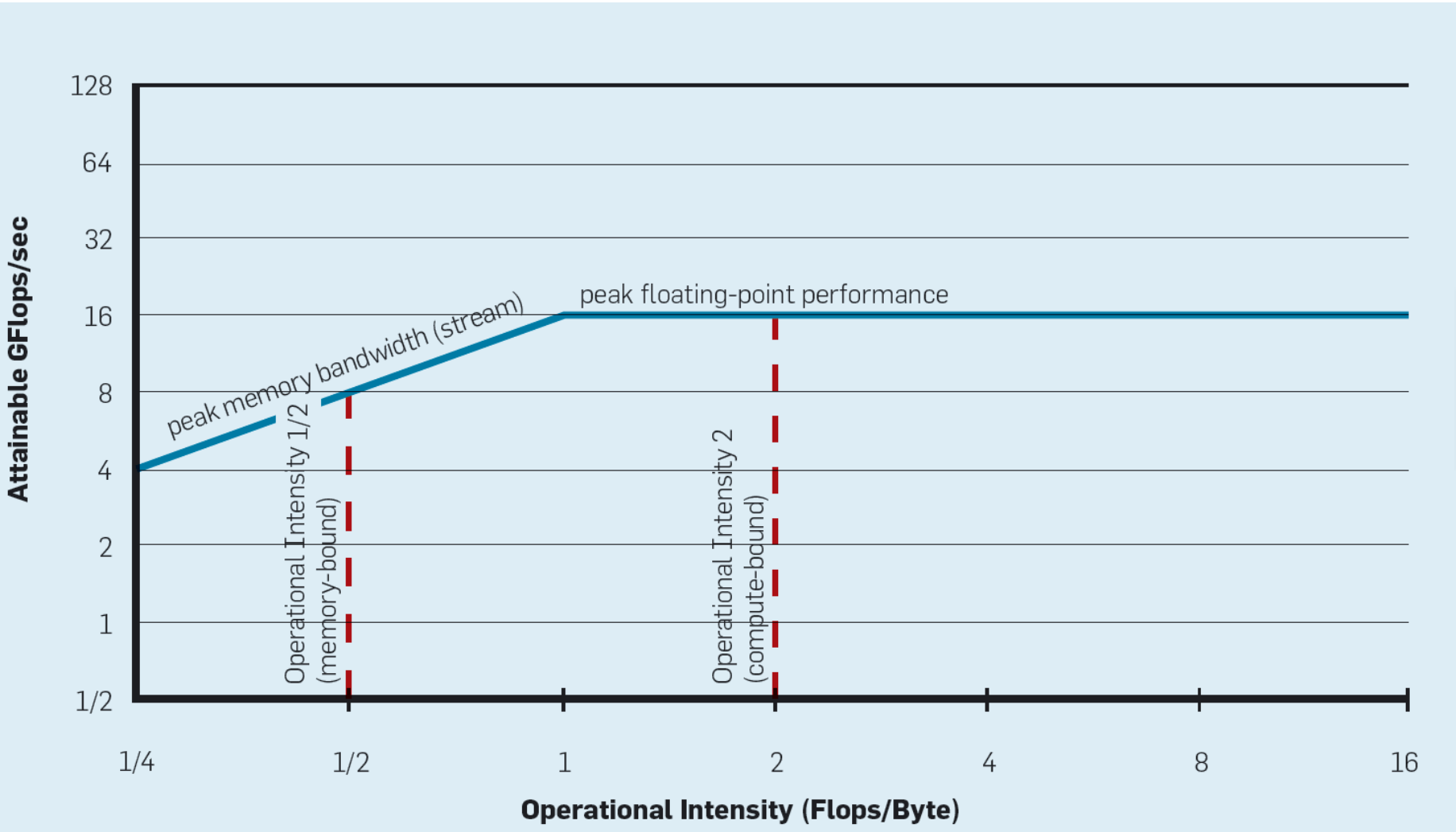
Bandwidth [GB/s] = memory bus frequency * bits per cycle * bus width

	Cores	Threads/ALUs	**GFLOPS**	**Bandwidth**
AMD HD 6970	384	1536	**2703**	**176**
AMD HD 7970	32	2048	**3789**	**264**
Intel Xeon Phi 7120	61	240	**2417**	**352**

What can go wrong ?!

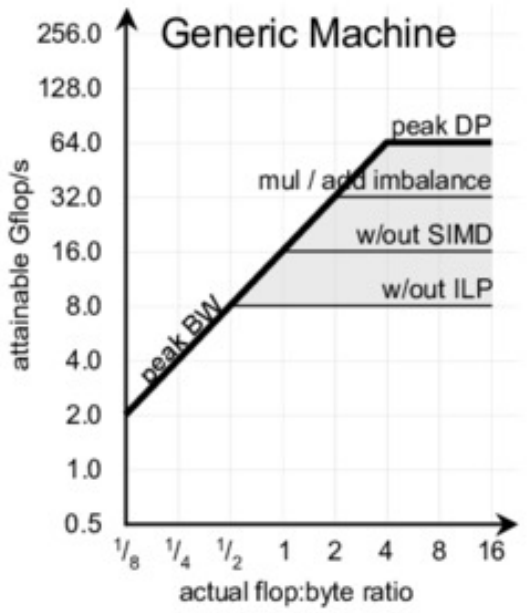
Alternatives?

The Roofline model: machine side



Multiple roofs … ?

- Peak performance
  - Using multiple cores
  - Using hyperthreading
  - Using vector units
  - Using FMA
  - Instruction mix correction

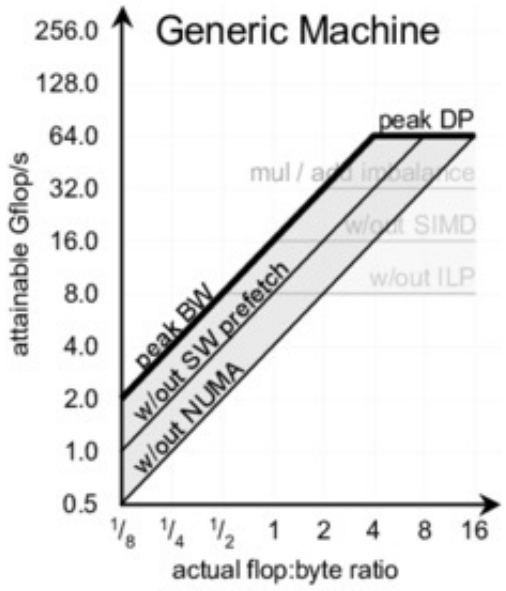


(a)

in-core parallelism ceilings

Multiple roofs … ?

- Memory performance
  - Caching (pushes ceiling up)
  - Lack of software prefetching (pushes ceiling down)
  - NUMA effects (may push ceiling down)



(b)
Bandwidth ceilings

Recap [2]

- Application performance metrics
  - Execution time
  - Cycle-based metrics
  - Derived metrics
    - FLOPs and alikes

Estimated performance

- Achieved (comp.) throughput: App_GFLOPs = #FLOPs / T
  - **Compute efficiency**: Ec = App_GFLOPs/peak * 100
- Achieved bandwidth: App_BW = #(RD+WR) [bytes] / T
  - **Bandwidth efficiency**: Ebw = App_BW/peak * 100
- Higher clock speed ?
- Different machine ?

Roofline model: application side

- Arithmetic Intensity = the number of arithmetic (floating point) operations per byte of memory that is accessed
  - Compute-bound?
    - i.e., adding more compute power increases performance.
  - Memory-bound?
    - i.e., adding more memory bandwidth increases performance.
- It is an application characteristic!
- To measure:
  - Count useful operations
  - Ignore “overheads”
    - Loop counters
    - Array index calculations
    - Branches

Last lecture's quiz

```
struct complex {float re; float im;};
int sum_array_3d(complex a[N][N][N])
{
    int i, j, k,
    float partial[N], sum = 0;
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++) {
          partial[i] += a[k][i][j].re
          sum += a[k][i][j].re;
        }
    return sum;
}
```

Throughput: 2xN^3 / 1s (FLOP/s)
Bandwidth: (4xN^3)*size(float) / 1s (B/s)

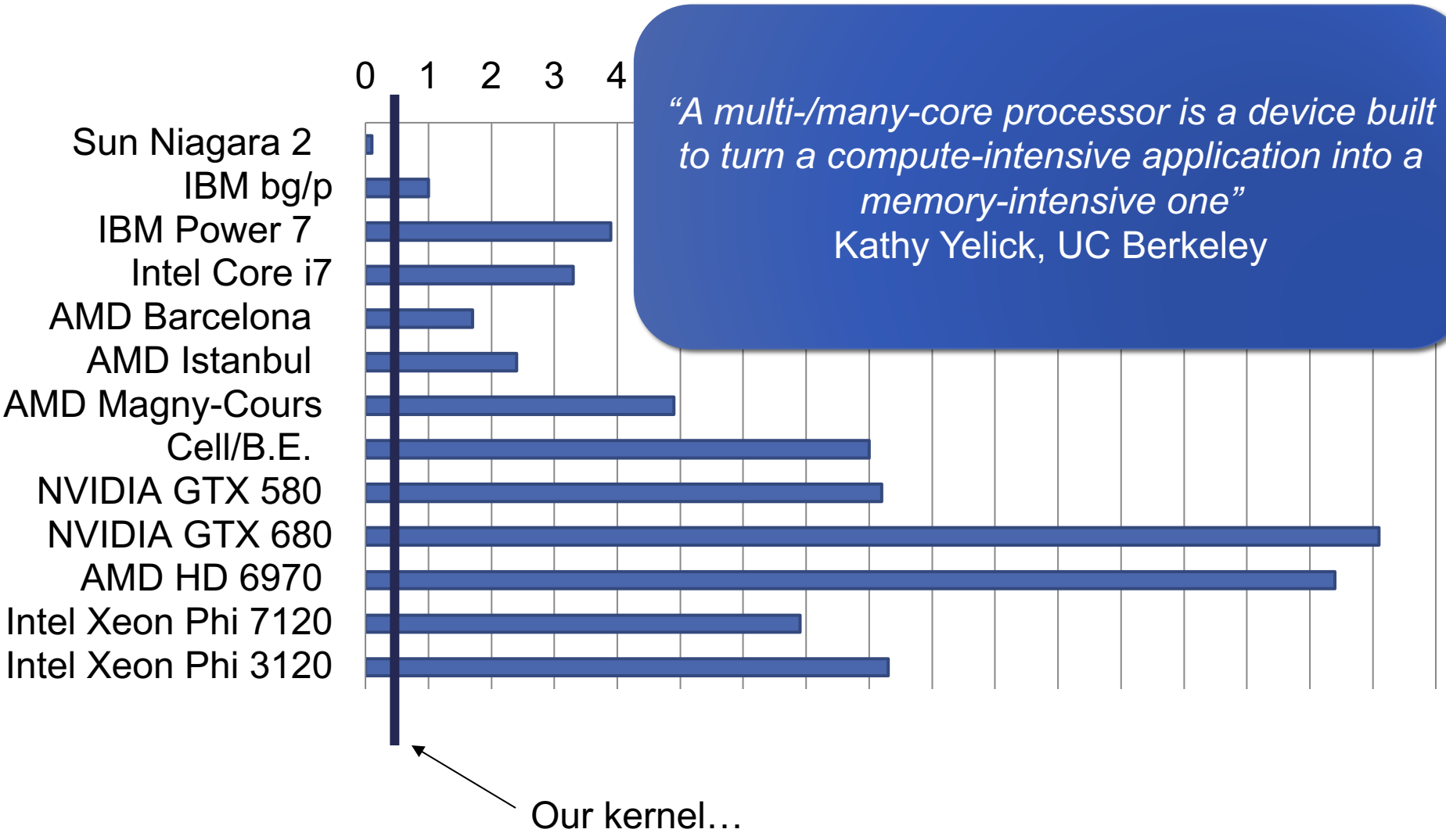
AI = … ~ 2/16

Many-core platforms

- The “balance” of a platform is defined as the peak performance/peak bandwidth
- Significance?

	Cores	Threads/ALUs	GFLOPS	Bandwidth	FLOPs/Byte
Intel Core i7	4	16	85	25.6	**3.3**
AMD Barcelona	4	8	37	21.4	**1.7**
AMD Istanbul	6	6	62.4	25.6	**2.4**
NVIDIA GTX 580	16	512	1581	192	**8.2**
NVIDIA GTX 680	8	1536	3090	192	**16.1**
AMD HD 6970	24	1536	2703	176	**15.4**
AMD HD 7970	32	2048	3789	264	**14.4**
Intel MIC 7120	**61**	**240**	**2417**	**352**	**6.9**

Compute or memory intensive?



"A multi-/many-core processor is a device built to turn a compute-intensive application into a memory-intensive one"
Kathy Yelick, UC Berkeley

Attainable performance

- Attainable GFlops/sec
  = min(Peak Floating-Point Performance,
    Peak Memory Bandwidth * Arithmetic Intensity)

Compute intensive

Memory intensive

- Peak iff AI_app ≥ PeakFLOPs / PeakBW
  - Compute-intensive iff AI_app ≥ (FLOPs/Byte)platform
  - Memory-intensive iff AI_app < (FLOPs/Byte)platform

Attainable performance

- Attainable GFlops/sec

  = min(Peak Floating-Point Performance,

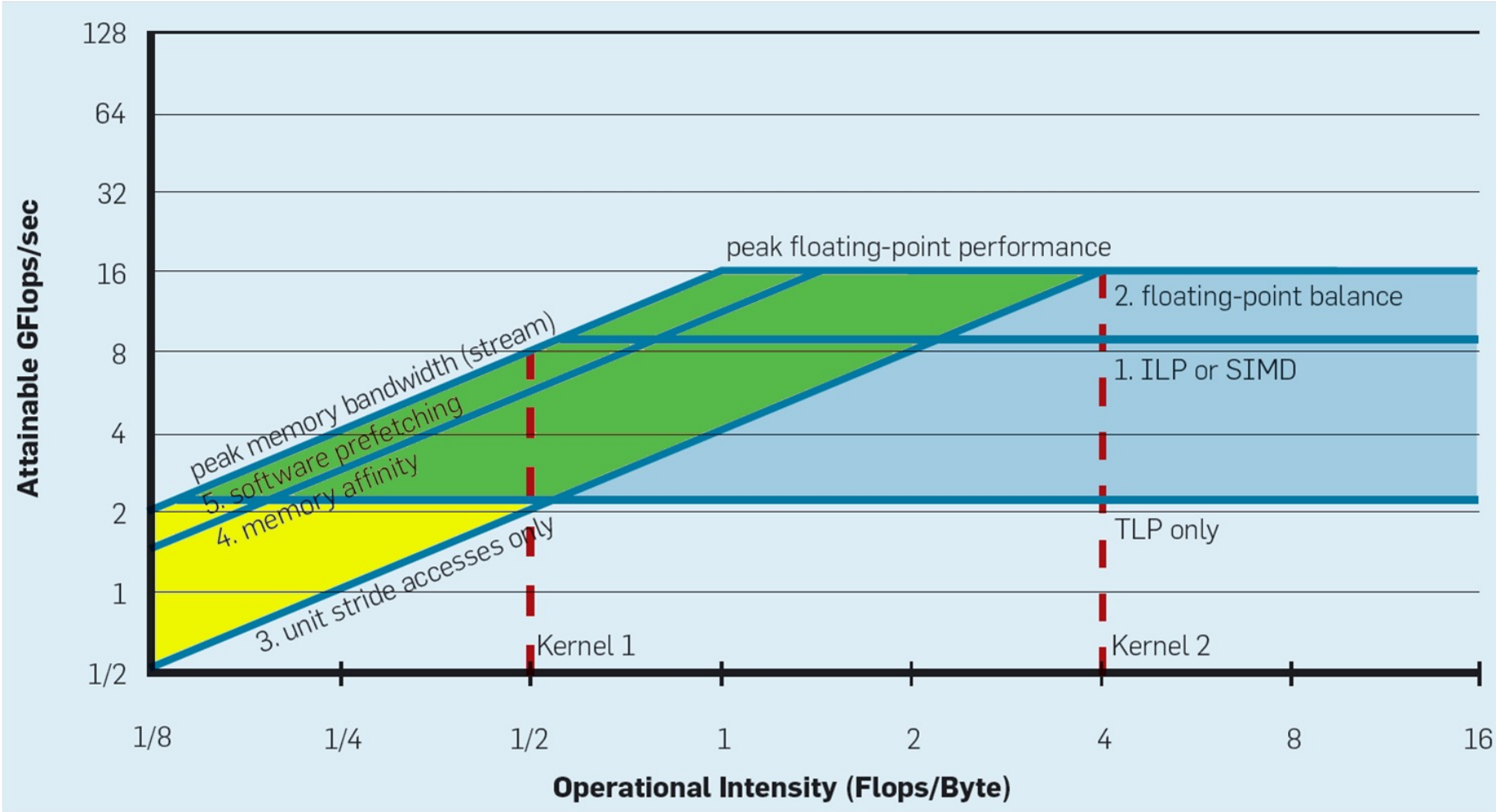
  Peak Memory Bandwidth * Arithmetic Intensity)

Compute intensive

Memory intensive

- Example: Quiz
  - AI = 3/8 ~ 0.4
  - NVIDIA GTX680
    - P = min (3090, 0,4 * 192) = 76,8 GFLOPs
      - Only **2.4%** of the peak
  - Intel MIC
    - P = min (2417, 0,4 * 352) = 140,8 GFLOPs
      - Only **5.8%** of the peak

Roofline: optimization regions



So … how does this work in practice?

Steps

- Calculate Roofline model for hardware
- Calculate AI for application
- Measure performance
  - Calculate GFLOPs
- Draw graph and analyse results
  - Check if HW models are correct
  - Check if results make sense
- Propose optimizations to address the observed bottleneck

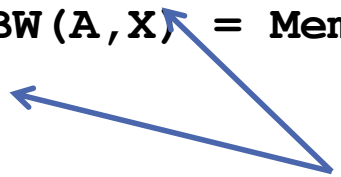
Example

Assume application A runs on platform X in **T(X,A)** (measured!) :

- **PeakCompute(X) = maxFLOP GFLOPS/s** (catalogue)
- **PeakBW(X) = maxBW GB/s** (catalogue)

- **RooflineCompute(A,X) = min(AI(A)*maxBW, maxFLOP)** (model)

- **AchievedCompute(A,X) = FLOP(A)/T(X,A)** (measurement)
- **AchievedBW(A,X) = MemOPs_in_Bytes(A)/T(X,A)** (measurement)



One of these is an “underestimate”, because T is either bound by memory or by operations.

- **UtilizationCompute = AchievedCompute(A,X)/PeakCompute(X)**
- **UtilizationBW = AchievedBW(A,X)/PeakBW(X)**

$Peak_{Compute}$ >= Roofline > $Achieved_{Compute}$
$Peak_{BW}$ > $Achieved_{BW}$

Use the Roofline model

- Determine what to do **first** to gain performance
  - Increase memory streaming rate
  - Apply in-core optimizations
  - Increase arithmetic intensity

- Read:

  Samuel Williams, Andrew Waterman, David Patterson

  “Roofline: an insightful visual performance model for multicore architectures”

Compute optimization(s)

Things to consider …

- "Classical optimizations"
  - Use cheaper operations and reduce redundant computation
- Improve instruction mix
  - Higher ILP => closer to peak
- Add SIMD
  - Higher ILP => closer to peak
- Use all cores
  - Use affinity to avoid thread migration

Notes:
1. High AI does not necessarily mean best performance!
2. High GFLOPs count does not necessarily mean peak performance.

Memory optimization(s)

Caches

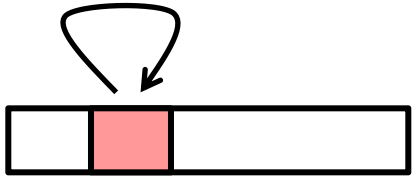
- *Cache:* A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- In a memory hierarchy:
  - For each k, the faster, smaller device at level k is a cache for the larger, slower device at level k+1.

- How do memory hierarchies work?
  - **Locality** => data at level k is used more often than data at level k+1.
    - Level k+1 can be slower, and thus larger and cheaper.

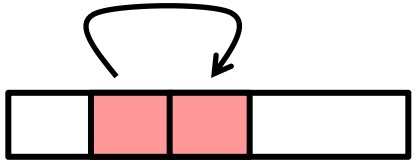
Why do caches work: Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time



Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data references
  - Reference array elements in succession (stride-1 reference pattern). — Spatial locality
  - Reference variable `sum` each iteration. — Temporal locality
- Instruction references
  - Reference instructions in sequence. — Spatial locality
  - Cycle through loop repeatedly. — Temporal locality

Reading data & caches

- Data read => block of data is determined based on address

- Block found in cache => hit !
  - Very fast access time
  - Move on …

- Block NOT found in cache => miss!
  - Miss penalty + read in the next cache
  - Decide where to place new data
  - Decide which block to remove (and potential update)
  - Do the actual data movement

Cache performance metric

- #Hits
  - Number of memory accesses successfully loaded from the cache
- Hit ratio
  - Percentage of memory accesses resulting in a hit
- Miss ratio
  - Percentage of memory accesses resulting in a miss
- Good locality => **high** hit ratio*
  - Except when other effects occur, like thrashing
  - Depends on the size and policy of the cache
- Bad locality => **low** hit ratio

"High" and "Low" are qualitative estimates => we'd rather use measurements…

Qualitative estimates of locality

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

- Quantitative estimate?
- … assuming 8 INT per cache line.

Qualitative estimates of locality

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

- Quantitative estimate?

… assuming 8 INT per cache line.

Qualitative estimates of locality

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M/2; i++)
        for (j = 0; j < N/8; j++)
            sum += a[i*2][j*8];
    return sum;
}
```

- Quantitative estimate?

… assuming 8 INT per cache line.

What about writes?

- Multiple copies of data exist:
  - L1, L2, L3, Main Memory, Disk
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- Typical
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

Caching and the Roofline model

What is the AI?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

1 op, 1 RD (4B)

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

1 op, 1 RD (4B)

We need to take caching into account!

- Hierarchical Roofline model
- Cache-aware Roofline model

Hierarchical Roofline

- Captures cache effects
- AI is Flop/Bytes after being filtered by lower cache levels
- Multiple Arithmetic Intensities (one per level of memory)
- AI dependent on problem size
- Memory/Cache/Locality effects are observed as decreased AI
- Requires performance counters or cache model/simulator to correctly measure AI

https://www.nersc.gov/assets/Uploads/2-SWillams-Roofline-Intro.pdf

Cache-aware Roofline

- Captures cache effects
- AI is Flop:Bytes as presented to the L1 cache
- Single Arithmetic Intensity
- AI independent of problem size
- Memory/Cache/Locality effects are observed as decreased performance
- Requires static analysis or binary instrumentation to measure AI

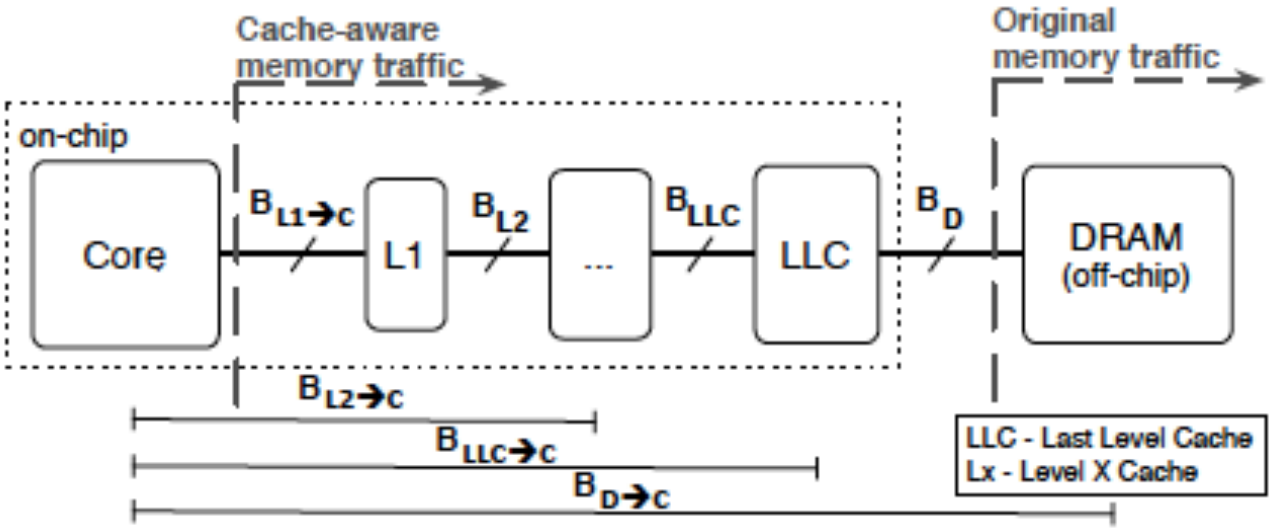
https://www.nersc.gov/assets/Uploads/2-SWillams-Roofline-Intro.pdf

Cache-aware Roofline Model*

Aleksandar Ilic et al. : “Cache-aware Roofline Model: Upgrading the Loft” and
“Beyond the Roofline: Cache-Aware Power and Energy-Efficiency Modeling for Multi-Cores”

Cache-aware Roofline model (CARM)*

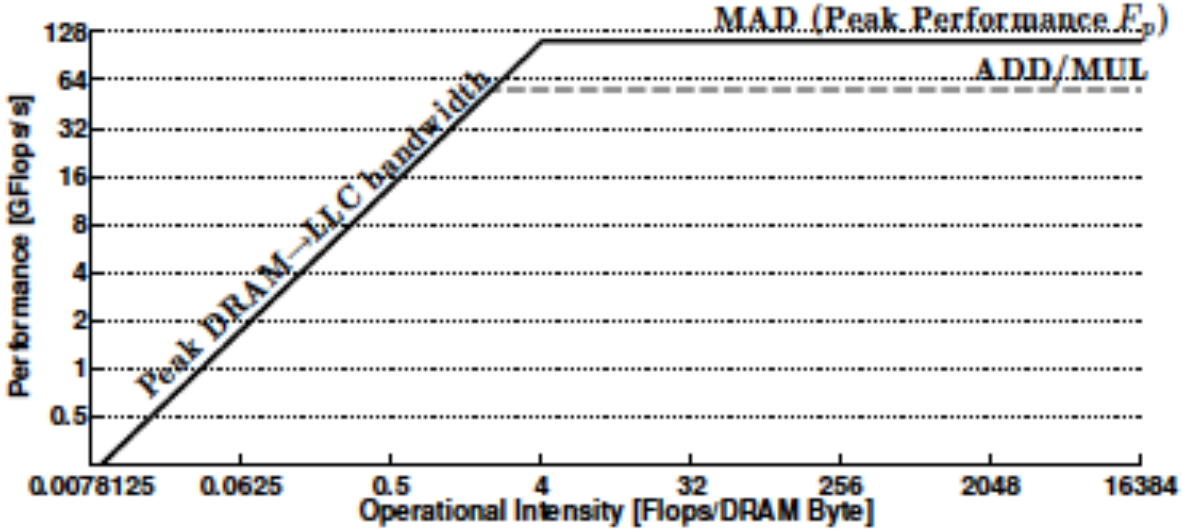
- Takes into account the memory hierarchy



- Subtle difference:
  - Original model: ONLY takes into account the traffic LLC-DRAM
  - New model: takes into account all traffic

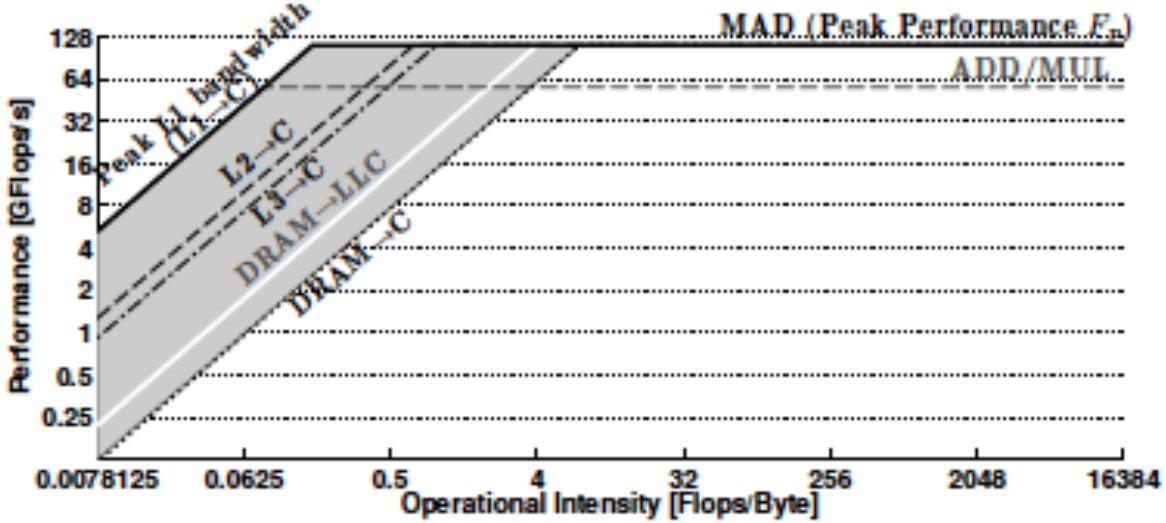
*Aleksandar Ilic – “Cache-aware Roofline model: upgrading the loft” (image source, too)

Cache-aware Roofline model (CARM)*



Original:

Takes into account traffic & bandwidths between memories



CARM:

All is seen from the perspective of the core. Thus, the bandwidths are adjusted to be from the core perspective.

Processor data

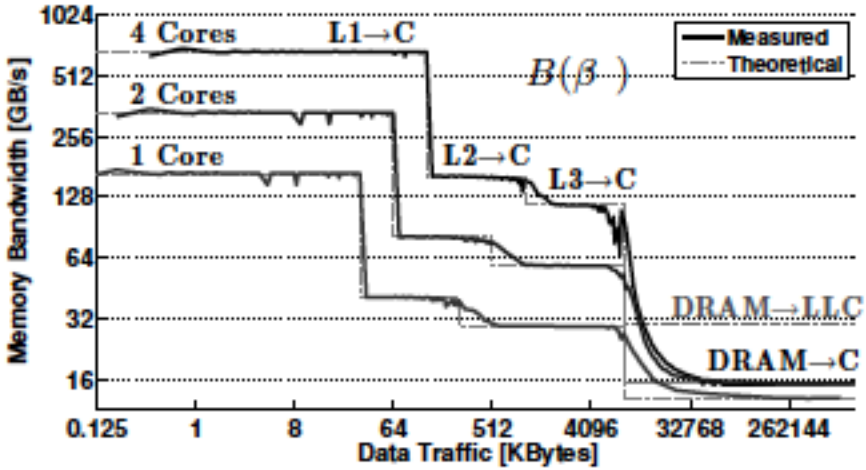
- Information from different CPUs

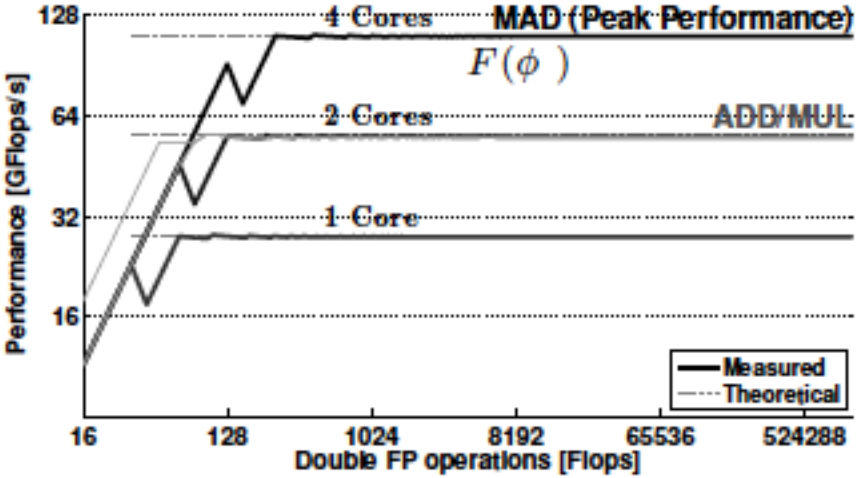
TABLE 1: General-purpose CPUs used for model evaluation

Intel processor model		2600K	3820	3930K	3770K
Architecture		Sandy Bridge	Sandy Bridge-E	Sandy Bridge-E	Ivy Bridge
#Cores		4	4	6	4
frequency [GHz]		3.4	3.6	3.2	3.5
F_p, AVX MAD (8 flops) [GFlops/s] (*)		108.8	115.2	153.6	112
L1 (per core)	size [bytes]	32K	32K	32K	32K
	bus width [bits]	384	384	384	384
	bandwidth [GB/s]	163.2	172.8	153.6	168
L2 size [bytes]		256K	256K	256K	256K
L3 size [bytes]		8M	10M	12M	8M
DRAM to LLC (DDR3)	#channels (8bytes/channel)	2	4	4	2
	bus frequency [MHz]	2×800	2×933	2×800	2×933
	bandwidth B_D [GB/s] (**)	25.6	59.7	51.2	29.9

(*) F_p = Max #flops × #Cores × frequency; (**) B_D = #channels × word size × bus frequency;

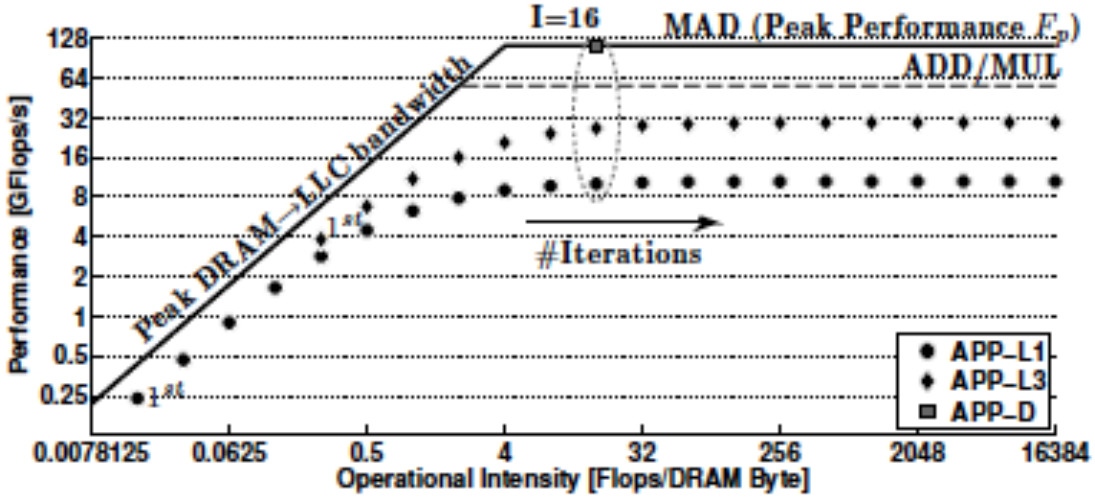
CARM: Roofs?

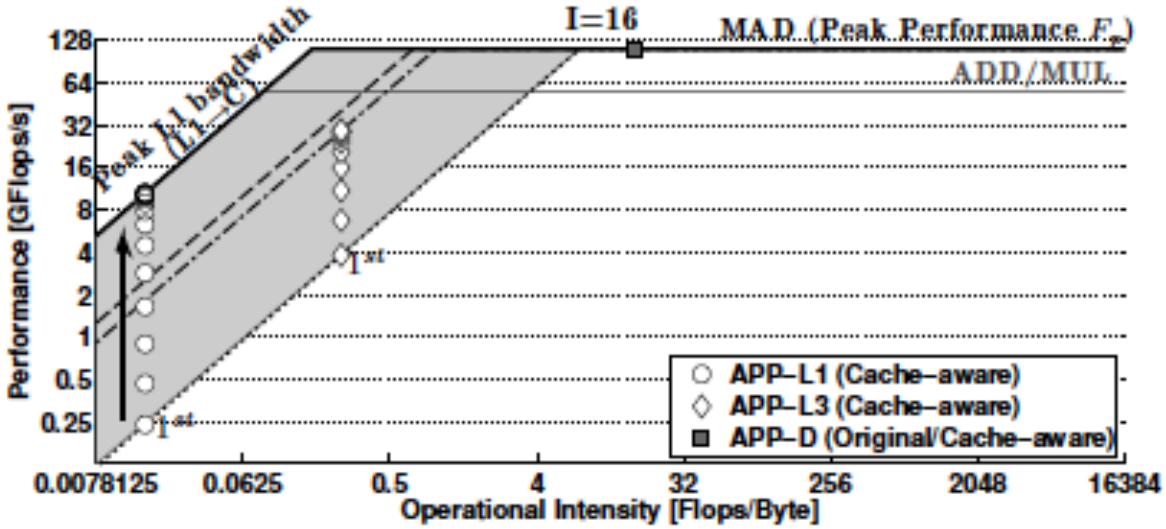




- Measuring the roofs
  - Microbenchmarking

CARM: How about applications?





For i=1..N

transfer B bytes;

compute O ops

APP-L1: dataset fits in L1.

Original:

N=1 => AI = O/B

N=2 => AI = O * 2 / B

=> AI changes !

CARM:

N=1 => AI = O/B

N=2 => AI = O * 2 / B * 2

=> AI is constant!

CARM: How about applications?

- AI is constant
  - APP-L1 is limited by L1 bandwidth => It can never be compute bound (as suggested by the original model!)
  - APP-L3 is limited by L3 bandwidth => It can still be improved if the memory operations are changes to use caches better
  - APP-D is the same in both cases, as it is limited by the DRAM bandwidth
    - AI is 16 all the time…