COSC 594: SCIENTIFIC COMPUTING FOR ENGINEERS

PERFORMANCE MODELING

Heike Jagode
jagode@icl.utk.edu

March 11, 2020
1. **Basic Essentials**
   - Abstract architecture model
   - Communication, Computation, and Locality
   - Arithmetic Intensity + Examples

2. **The Roofline Model**

1. **Refinement of the Roofline Model: Ceilings**
   - Bandwidth ceilings
   - In-Core ceilings
   - Arithmetic Intensity Walls
LEARNING OBJECTIVES

- Define **abstract architectural model** used by Roofline model (RM)
  - Understanding this model is critical in order to apply RM to wide variety of computational kernels

- Introduce concept of **arithmetic intensity**
  - Hands-on examples

After completing this lecture, you should be able to:
- Describe and quantify the main factors affecting single-node performance
- Construct analytical performance models for simple sequential kernels
- Validate and refine performance models using benchmarks
HIGH-LEVEL OVERVIEW: THE ROOFLINE MODEL

• Is a visually intuitive performance model

• Designed to drive programmers towards better understanding of performance on modern computer architectures
  → not only provides programmers with realistic performance expectations
  → also specifies potential challenges to performance

• Knowledge of these bottlenecks drive programmers to implement particular classes of optimizations

• Focus on architecture-oriented roofline models (not using performance counters)
RM presumes simple architectural model consisting of black boxed:

- **computational elements** (e.g. ________________________________)
- **memory elements** (e.g. ________________________________)

interconnected by a network
RM presumes simple architectural model consisting of black boxed:

- **computational elements** (e.g. ____________________________)
- **memory elements** (e.g. ____________________________)

interconnected by a network

- Whenever one or more comp elements may access a mem element, that memory element is considered **SHARED**
RM presumes simple architectural model consisting of black boxed:

- **computational elements** (e.g. ____________________________)
- **memory elements** (e.g. ____________________________)

interconnected by a network

- Whenever one or more comp elements may access a mem element, that memory element is considered **SHARED**
- No restriction on the # of elements
- No restriction on balance of comp and mem elements
- At any given level of hierarchy, comp elements may only communicate either with mem elements at that level, or with mem elements at coarser level
Abstract Architecture Model 1/2

- RM presumes simple architectural model consisting of black boxed:
  - **computational elements** (e.g. ____________________________)
  - **memory elements** (e.g. _________________________________)  
    interconnected by a network
- Whenever one or more comp elements may access a mem element,  
  that memory element is considered **SHARED**
- No restriction on the # of elements
- No restriction on balance of comp and mem elements
- At any given level of hierarchy, comp elements may only communicate  
  either with mem elements at that level, or with mem elements at coarser  
  level
  - large # of possible topologies/hierarchies exist
  - allowing RM to be applied to a large # of current and future  
    computer architectures
Abstract Architecture Model 2/2

2 different dual processor architectures
→ any proc can reference any memory location

(a) Distributed Memory: processors connected to separate memories
   (any processor can access any DRAM location)

(b) Processors connected to a common Shared Memory

NOTE: arrows denote the architecture's ability to access information
   (not necessarily hardware connectivity)
COMMUNICATION, COMPUTATION, AND LOCALITY

Communication:

- A kernel can be expressed as movement of data from one or more memories to a processor where it may be buffered, duplicated or computed on.
- That modified data or any new data is then moved back to memory.

→ Communication is bounded by characteristics of processor-memory interconnect.
(There is a maximum bandwidth on any link as well as a max bandwidth limit on any subset of links *)
**COMMUNICATION, COMPUTATION, AND LOCALITY**

**Communication:**
- A kernel can be expressed as movement of data from one or more memories to a processor where it may be buffered, duplicated or computed on.
- That modified data or any new data is then moved back to memory.
- Communication is bounded by characteristics of processor-memory interconnect.
  (There is a maximum bandwidth on any link as well as a max bandwidth limit on any subset of links *)

**Computation:** (In this lecture consists of FP Ops including multiply, add, compare, etc.)
- Each proc has an associated comp rate.
- We won’t analyze how perf is distributed among cores within a multicore chip. (only when using a hierarchical model for multi-cores)
Communication:

• A kernel can be expressed as movement of data from one or more memories to a processor where it may be buffered, duplicated or computed on.

• That modified data or any new data is then moved back to memory.

→ Communication is bounded by characteristics of processor-memory interconnect.

(There is a maximum bandwidth on any link as well as a max bandwidth limit on any subset of links)

Computation: (In this lecture consists of FP Ops including multiply, add, compare, etc.)

• Each proc has an associated comp rate.

• We won’t analyze how perf is distributed among cores within a multicore chip. (only when using a hierarchical model for multi-cores)

Locality:

• Some initial locality of data in memory \(_i\) – once moved to proc \(_j\) we may assume that caches provide for locality within the processor.

→ subsequent ref. to that data will hit.
**ARITHMETIC INTENSITY (AI)**

- Is a kernel’s **ratio of computation to traffic** (measured in flops/bytes)
- Note: traffic is the **volume of data** to/from memory
  It is not # of loads and stores

E.g. Processors whose caches filter most memory requests have high AI

**Machine Balance (MB):**

- Similar concept which represents the **ratio of peak FP perf to peak bandwidth**

  → A simple comparison between MB and AI may provide insight for potential bottlenecks
    - E.g. If AI of a kernel exceeds MB, it is likely to spend more time in ? than ? → kernel is ?-bound!

- Very simple approximation that covers up many details of computer architecture → results in perf far below perf expectations
  → Situations like this motivated the creation of the RM
Hands-On In Class:

1. Example
1. Example of AI: Vector Magnitude

```c
temp = 0.0
for ( i=0; i<N; i++ ) {
    temp += A[i]*A[i]
}
magnitude = sqrt(temp);
```

- Assume all arrays are double precision
- Assume N is sufficiently large → array does not fit in cache

Recap: AI is ratio of total FP operations to total DRAM bytes

FLOPs = 
Traffic = 
→ AI =
HANDS-ON IN CLASS:

2. EXAMPLE
2. Example of AI: Stencil Sweep of 2D PDE

```c
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        C[i][j] = a*A[i][j] +
                   b*(A[i][j-1] +
                      A[i-1][j] +
                      A[i+1][j] +
                      A[i][j+1]);
    }
}
```

- Assume all arrays are **double precision**
- Assume: \(16N \leq \text{cache size} < 16N^2\)
- Note, write-allocate cache architectures generate a **read** for initial fill and a **write** for the actual write back.

\[
\text{FLOPs} = \text{Traffic} = \rightarrow AI = \]

---

DICOM, TEER, ICL, UTR

3/10/20
HANDS-ON IN CLASS OR HOMEWORK:

3. EXAMPLE
3. **Example of AI: Dense Matrix-Matrix Multiply**

```java
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        C[i][j] = 0.0;
        for (k=0; k<N; k++) {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

- Assume all arrays are **double precision**
- Assume: **cache size > 24N²** \(\rightarrow\) \(A[i][j], B[i][j], C[i][j]\) are kept in cache and only the initial write back reference will generate DRAM traffic.

\[\text{FLOPs} = \text{Traffic} = \Rightarrow \text{AI} = \]
1. **Basic Essentials**
   - Abstract architecture model
   - Communication, Computation, and Locality
   - Arithmetic Intensity + Examples

2. **The Roofline Model**

   1. **Refinement of the Roofline Model: Ceilings**
      - Bandwidth ceilings
      - In-Core ceilings
      - Arithmetic Intensity Walls
Recap: What we covered so far

- abstract architectural model + comm, comp, locality
- kernel’s estimated Arithmetic Intensity

Now, ready for the “Roofline Model”

- Allows programmers to determine attainable performance
- Considers the two principle performance bounds in isolation:
  - ?? and ??
- Compares their corresponding times to determine
  - bottlenecks as well as
  - attainable performance
Consider a simple kernel that runs on machine (b) and that must:
(a) transfer $B$ Bytes of data from mem$_0$
(b) perform $F/2$ FLOPs on both CPUs

Suppose:
(c) memory can support $\text{PeakBandwidth}$ Bytes/sec
(d) The two CPUs combined can perform $\text{PeakPerformance}$ FLOPs/sec
THE ROOFLINE

• How long will it take to transfer the data?

• How long will it take to compute on the data?

• Suppose, one may perfectly overlap comm and comp. How long will it take to finish the kernel until completion?
After computing the reciprocal and multiplying by F flops, we observe performance is bound to:

\[
\text{Attainable Performance (GFlops/s)} = \min \left\{ \frac{\text{Peak Performance}}{\text{Peak Bandwidth} \times \text{AI}} \right\}
\]

Where \( \text{Arithmetic Intensity (AI)} = \frac{F}{B} \)

**Question:**

A given architecture has a fixed peak bandwidth and peak performance. Is the AI fixed as well when running different kernels on this given architecture?
This implies, can plot **Attainable Performance** as a function of AI

**Example:**

- 2.3 GHz dual-socket x quad-core Opteron 2356
- Determine Max Bandwidth using the STREAM benchmark: 16.6 GB/s
- Using a processor optimization manual: Max Perf = 73.6 Gflops/s

**Questions:**

Is it possible to always attain both?
• 2.3 GHz dual-socket x quad-core Opteron 2356
• Determine Max Bandwidth using the STREAM benchmark: 16.6 GB/s
• Using a processor optimization manual: Max Perf = 73.6 Gflops/s

Figure (log-log scale) visualizes Attainable Performance equation for SMP Opteron example
Why STREAM benchmark to determine Max Bandwidth?

- Peak Bandwidth usually unachievable
  - Max bandwidth often not available in both directions
  - Protocol overhead
  - Latencies that cannot be hidden by prefetching

- STREAM Benchmark measures effective peak memory bandwidth
BENCHMARKS

- **Low-Level Benchmark**
  - Measures one characteristic of a machine
  - Example 1: STREAM memory bandwidth benchmark

- **Kernel Benchmark**
  - Measure performance of a single application kernel that performs just one type of computation
  - Example: Polybench kernel benchmarks

- **Full Application Benchmarks**
  - Used for system procurement and acceptance
  - Not useful for performance modeling
  - Example: DoD Technology Insertion Benchmark Suite (TI-xx)
BACK TO: THE ROOFLINE EXAMPLE

- 2.3 GHz dual-socket x quad-core Opteron 2356
- Determine Max Bandwidth using the STREAM benchmark: 16.6 GB/s
- Using a processor optimization manual: Max Perf = 73.6 Gflops/s

Figure (log-log scale) visualizes Attainable Performance equation for SMP Opteron example
Given $\text{Flops}_{\text{max}}$ and $\text{BW}_{\text{max}}$, where is the cross-over point between memory-bound and CPU-bound applications?

Note, **slope** of the roofline in the BW-limited region is actually the machine’s STREAM bandwidth

On log-log scale, line always appears at a 45-degree angle (on this scale, if doubling the BW will shift the line up instead of changing its perceived slope)

As AI increases, so does the performance bound

However, at the machine’s flop:byte ratio, performance bound saturates at the machine’s peak performance

Model can be used to bound the Opteron’s **Attainable Performance** for variety of computational kernels

**Kernel 1**’s AI = 1 $\rightarrow$ is in BW-limited region (i.e. perf is still increasing with AI)

**Kernel 3**’s AI = 16 $\rightarrow$ kernel is ultimately compute-bound

**Kernel 2**’s perf heavily depends on exact AI; the kernel’s and machine’s ability to perfectly overlap communication and computation.
Consider the following kernel:

```c
for ( i=0; i<N; i++) {
    A[i] = B[i]+C[i]*D[i]
}
```

Assume a 3.0 GHz processor that can execute 4 flops/cycle and that has a peak memory bandwidth of 10.6 Gbytes/s.

**Question:**
What percentage of peak would you expect the above kernel to obtain?
THE ROOFLINE: SUMMARY

• Performance is no longer a scalar 
  → but a coordinate in AI-Gflops/s space

• Roofline itself is only a performance bound (upper bound)
  • In practice, it’s common that actual perf will be below the roofline
  • It can never be above roofline

• Programmers are interested in both: architectural analysis and program optimization
  → Necessary to understand WHY perf is below and not on roofline
  → How can we optimize a kernel/application to correct this?

→ Roofline model can be refined to enhance its utility
1. **Basic Essentials**
   - Abstract architecture model
   - Communication, Computation, and Locality
   - Arithmetic Intensity + Examples

2. **The Roofline Model**

   1. **Refinement of the Roofline Model: Ceilings**
      - Bandwidth ceilings
      - In-Core ceilings
      - Arithmetic Intensity Walls
Architectures exploit a number of techniques to hide memory latency
- HW, SW prefetching, TLB misses
- And increase memory bandwidth
  - Multiple controllers, burst-mode accesses, NUMA

For each of architectural paradigms, proportional set of optimizations that must be implemented to extract peak memory performance

Focus: enumerate these performance obstacles
- Visualize them as Bandwidth Performance Ceilings
- A ceiling is comparable to the roofline: it denotes complete failure to exploit an architectural paradigm
- SW optimization removes these ceilings
NUMA 
(Non-Uniform Memory Access)

- Memory design for multiprocessing
- Memory access time depends on mem location relative to processor
  - Proc can access its local mem faster than non-local or shared mem

Loop-Example:
- Initialize values
- Distributed data among proc via OpenMP
- Virtual add. appears contiguous but physical add. mapped to mem on different sockets

Optimized case:
- When proc compute on data, pieces of array are in mem to which they have highest BW
NUMA
(NON-UNIFORM MEMORY ACCESS)

- Supposed OpenMP pragmas are omitted
  → Likely that entire array placed in one memory

Disadvantages:
- Give up half of system’s peak BW by not using other memory
- Loose additional performance as link 1 likely has lower BW (0, 3) but must transfer as much data

```c
for (j=0; j<N; j++){
    a[j] = 1.0;
    b[j] = 2.0;
    c[j] = 0.0;
}
```
NUMA (Non-Uniform Memory Access)

- Plot resultant BW on roofline model
- 2.5x performance degradation
  → depresses performance for all memory-bound kernels
  → expands range of memory-bound AI to ~10 flops/byte
  → Those perf issues extremely difficult to detect regardless of OpenMP, POSIX threads, other threading library
NUMA (Non-Uniform Memory Access)

- Plot resultant BW on roofline model
- 2.5x performance degradation
  → depresses performance for all memory-bound kernels
  → expands range of memory-bound AI to ~10 flops/byte
- Those perf issues extremely difficult to detect regardless of OpenMP, POSIX threads, other threading library

One should strongly consider only threading applications within a socket instead of across an entire SMP node !!!
Modern micro-processors use virtual memory
• Accelerate the translation to physical addresses via small highly-associative translation lookaside buffers (TLBs)
→ Unfortunately, TLBs act like caches on a page table
→ If a kernel’s working set (measured in page table entries) exceeds TLB capacity → TLB misses
• Happens more often than you might think!!
→ Ceiling: Implement a version of STREAM that scales # of streams for operations like e.g. TRIAD

```
#pragma parallel for
for (i =0; i<N; i++) {
    a[i] = b[i] + c[i] * SCALAR;
}
```

• This would often result in about the same BW for low number of steams, but would suddenly dip for an additional array
• Plot dip on Roofline model as a bandwidth ceiling
Modern core architectures are complex
→ floating-point performance is not simply a function of AI alone
→ Instead, arch. exploit number of paradigms to improve peak perf.
  • Pipelining
  • Superscalar out-of-order execution
  • SIMD
  • HW multi-threading
  • Multi-core … etc.
→ For each of architectural paradigms, proportional set of optimizations that must be implemented to extract peak performance (either by compiler or user)

**Focus**: enumerate these performance obstacles
**Visualize** them as **In-Core Performance Ceilings**
INSTRUCTION-LEVEL PARALLELISM

- Every instruction on every arch. has an associated latency
  - Time from when instruction’s operands are available to time where the results available to other instructions (assuming no other resource stalls)
  - For FP instr. like ADD or MUL, latencies small (< 8 cycles)
- Also, processors have associated dispatch rate (BW) that represents number of independent instructions that can be executed per cycle

→ **Instruction-Level parallelism**: measure of how many of the operations in a program can be performed simultaneously

**Example:**

\[
e = a + b \\
f = c + d \\
g = e \times f
\]

**Question:** Can all three operations be calculated at the same time?
INSTRUCTION-LEVEL PARALLELISM

• Every instruction on every arch. has associated latency
  • Time from when instruction’s operands are available to time where
    the results available to other instructions
    (assuming no other resource stalls)
  • For FP instr. like ADD or MUL, latencies small (< 8 cycles)

• Also, processors have associated dispatch rate (BW) that represents
  number of independent instructions that can be executed per cycle

→ Instruction-Level parallelism: measure of how many of the operations
  in a program can be performed simultaneously

Example:

```
e = a + b
f = c + d
g = e * f
```

Question: Can all three operations be calculated at the same time?

3. Operation depends on 1. and 2; But 1. and 2. can be calculated in parallel
   Each op. calculated in one unit of time → all can be calc. in two → ILP = 3/2
INSTRUCTION-LEVEL PARALLELISM

• Every instruction on every arch. has associated latency
  • Time from when instruction’s operands are available to time where
    the results available to other instructions
    (assuming no other resource stalls)
  • For FP instr. like ADD or MUL, latencies small (< 8 cycles)

• Also, processors have associated dispatch rate (BW) that represents
  number of independent instructions that can be executed per cycle

→ Instruction-Level parallelism: measure of how many of the operations
  in a program can be performed simultaneously

Example: e = a + b
f = c + d

3. Operation depends on 1. and 2; But 1. and 2. can be calculated in parallel
Each op. calculated in one unit of time → all can be calc. in two → ILP = 3/2

When a thread of execution falls short of expressing this degree of parallelism, functional units will go idle, and performance will suffer !!!
INSTRUCTION-LEVEL PARALLELISM

• Add performance ceilings as result of insufficient ILP

• Without ILP, AI at which a processor becomes compute-bound is much lower

→ it may be possible that many kernels show signs of being compute-bound, yet deliver poor performance
(a) Loop is naively unrolled (either by user or compiler)
(b) Loop with computation of partial sums

**Question**: Which of the two loops increased ILP? (double-precision)
ILP: EXAMPLE

(a) Loop is naïvely unrolled (either by user or compiler)
(b) Loop with computation of partial sums

**Question**: Which of the two loops increased ILP? *(double-precision)*

(a) FP-ILP remains constant:

Adds to sum must be serialized

ADDS are *commutative*: \(a + b = b + a\)
But not necessarily *associative*: \((a + b) + c \neq a + (b + c)\)
(a) Loop is naïvely unrolled (either by user or compiler)
(b) Loop with computation of partial sums

**Question:** Which of the two loops increased ILP? *(double-precision)*

(a) FP-ILP remains constant:

```
for(i=...){
    sum+=a*b[i];
}
```

Adds to sum must be serialized

ADDs are *commutative*: \(a+b = b+a\)
But not necessarily *associative*: \((a+b)+c \neq a+(b+c)\)

(b) FP-ILP increases with unrolling

```
for(i=...){
    sum0+=a*b[i];
    sum1+=a*b[i+1];
}
```

AI will be sufficiently greater

\(\Rightarrow (b) \text{ should outperform (a)}\)
FUNCTIONAL UNIT HETEROGENEITY

- Modern processors (AMD Opteron, Intel Nehalem, SandyBridge, IvyBridge) have FP-XU opt. for certain instructions
  - Two pipelines capable to simultaneously execute two FP inst.
  - One pipeline only performs FP \texttt{adds}, other only FP \texttt{muls}

Question:
- How may this affect performance of codes that are dominated by one or the other instruction?
FUNCTIONAL UNIT HETEROGENEITY

- Modern processors (AMD Opteron, Intel Nehalem, SandyBridge, IvyBridge) have FP-XU opt. for certain instructions.
  - Two pipelines capable to simultaneously execute two FP inst.
  - One pipeline only performs FP **adds**, other only FP **muls**

**Question:**
- How may this affect performance of codes that are dominated by one or the other instruction?

**Example:** Stencil Sweep of 2D PDE **VS.** MM Multiply

```c
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        C[i][j] = a*A[i][j] +
                  b*(A[i][j-1] + A[i-1][j] +
                    A[i+1][j] + A[i][j+1]);
    }
}
```

**VS.**

```c
C[i][j]=0.0;
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        for (k=0; k<N; k++) {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```
FUNCTIONAL UNIT HETEROGENEITY

- Modern processors (AMD Opteron, Intel Nehalem, SandyBridge, IvyBridge) have FP-XU opt. for certain instructions
  - Two pipelines capable to simultaneously execute two FP inst.
  - One pipeline only performs FP adds, other only FP muls

**Question:**
- How may this affect performance of codes that are dominated by one or the other instruction?

→ For codes that are dominated by one or the other, attainable perf. will be half vs. code with perfect balance between muls, adds

**Example:** Stencil Sweep of 2D PDE VS. MM Multiply

```c
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
    }
}
```

```c
C[i][j]=0.0;
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        for (k=0; k<N; k++) {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```
FUNCTIONAL UNIT HETEROGENEITY

- Can create series of ceilings based on ratio of \texttt{adds} and \texttt{muls}
- As ratio gets further away from 1, ceiling approaches $\frac{1}{2}$ of peak

**Fused Multiply-Add (FDA):**
- Instr. performs \texttt{muls} and \texttt{adds} in sequence instead of parallel
- Execute same # of FP ops as machine with 2x instruction issue width

**Question:**
- How does FMA affect performance? Similar ceilings like with MAD or not?
Can create series of ceilings based on ratio of \textit{adds} and \textit{muls}.

As ratio gets further away from 1, ceiling approaches $\frac{1}{2}$ of peak.

**Fused Multiply-Add (FDA):**

- Instr. performs \textit{muls} and \textit{adds} in sequence instead of parallel.
- Execute same \# of FP ops as machine with 2x instruction issue width.

**Question:**

- How does FMA affect performance? Similar ceilings like with MAD or not?

\(\rightarrow\) FMA creates similar performance issue unless code is dominated by FMAs \(\rightarrow\) performance may drop by factor of two!
DATA-LEVEL PARALLELISM (SIMD)

- With single instruction, program may express two- or four-way DLP

**Example:**
- X86 instruction `addps` performs 4 SP FP add operations in parallel
- Ideally, compilers should recognize this form of parallelism + generate these instr.
- Unfortunately, compilers often fail (due to the implementation’s rigid nature)
- Failure to exploit these instr. can substantially depress performance
**SIMD: Example**

- Substantial ILP, but only FP adds are performed → no DLP

- Replace C assignm. with compiler **scalar** form of intrinsics (small fct. mapped to 1 or 2 instr.)
  → still no increase in degree of DLP
  → Perf bound not improved

- Replace scalar form with **_pd** form of intrinsics

- Should unroll loop eight times → express both 2-way DLP + 4-way ILP
Example of Ceilings associated with data-level parallelism

1. loop: no DLP  → performance bound to <18.4 Glops/s
3. loop: 2-way DLP  → performance bound improved to 36.8 Gflops/s
COMBINING CEILINGS

- All ceiling independent → may be combined as needed
- E.g lack of ILP can be combined with lack of DLP → severely depressed performance
→ Can add multiple ceilings (representing lack of different forms of parallelism) on single roofline

Question:
→ But what’s the order of multiple different ceilings?
- Use intuition based on (always bottom-up):
  - What is most likely implicit in algorithm?
  - What is a compiler going to discover/provide?
  - What can a programmer provide?
  - What can never be exploited for this kernel (those are the ceilings placed near roofline)
Based on this intuition, any of the 3 equivalent roofline models can be used.
Recap:

- Up until now, assumed total DRAM bytes within AI ratio is dominated by “compulsory” memory traffic
  
  \[ \text{AI} = \text{kernel’s ratio of computation to traffic} \] (in flops/bytes)
  
  - BUT on real codes, a # of other significant terms in the AI denominator
  - Accurately determining the true flop:DRAM bytes ratio
  
  - Implies taking the 3Cs into consideration:
    - **Compulsory misses**: misses caused by first reference to a location in memory
    - **Capacity misses**: occur regardless of associativity or block size, solely due to the finite size of the cache
    - **Conflict misses**: misses that could have been avoided, had the cache not evicted an entry earlier
Arithmetic Intensity Wall

\[ AI_{\text{new}} = \frac{\text{Total FP Operations}}{\text{Compulsory} + \text{Allocation} + \text{Capacity} + \text{Conflict Memory Traffic} + \ldots} \]

- Write allocation traffic, capacity cache misses, conflict cache misses, etc. contribute to reduced AI
- Walls constrain AI, and when BW-limited \( \rightarrow \) constrain performance

Capacity + conflict misses depend on problem size and if it exceeds cache capacity + associativity
\( \rightarrow \) Walls execution-dependent rather than architecture dependent

May see performance degradation for small or big problem sizes?
**Arithmetic Intensity Wall**

\[
AI_{\text{new}} = \frac{\text{Total FP Operations}}{\text{Compulsory} + \text{Allocation} + \text{Capacity} + \text{Conflict Memory Traffic} + \ldots}
\]

- Write allocation traffic, capacity cache misses, conflict cache misses, etc. contribute to reduced AI.
- Walls constrain AI, and when BW-limited \(\rightarrow\) constrain performance.

Capacity + conflict misses depend on problem size and if it exceeds cache capacity + associativity
\[\rightarrow\] Walls execution-dependent rather than architecture dependent.

May see performance degradation for small or big problem sizes?

Big problem sizes may result in substantial perf. loss due to capacity and conflict miss traffic!
COMPULSORY MISS TRAFFIC

- Calculation: straightforward
- Note, compulsory traffic != min memory traffic of algorithm
- compulsory traffic = min memory traffic required for particular impl.

Question:
What’s the most obvious example of elimination of compulsory traffic?

```c
for ( i=0; i<N; i++) {
    temp += A[i]*A[i]
}
```

Traffic =
COMPULSORY MISS TRAFFIC

• Calculation: straightforward
• Note, compulsory traffic != min memory traffic of algorithm
• compulsory traffic = min memory traffic required for particular impl.

Question:
What’s the most obvious example of elimination of compulsory traffic?

```c
for ( i=0; i<N; i++) {
    temp += A[i]*A[i]
}
```

Traffic = 8*N Bypes \( (N \text{ doubles}) \) \( \text{ (Note, 2^{nd} access to A[i] exploits cache/register locality)} \)
COMPULSORY MISS TRAFFIC

- Calculation: straightforward
- Note, compulsory traffic != min memory traffic of algorithm
- compulsory traffic = min memory traffic required for particular impl.

Question:
What's the most obvious example of elimination of compulsory traffic?

```c
for ( i=0; i<N; i++) {
    temp += A[i]*A[i]
}
```

Traffic = 8*N Bypes (N doubles) (Note, 2\textsuperscript{nd} access to A[i] exploits cache/register locality)

→ Changing data types
e.g. \texttt{double} to \texttt{single} precision or \texttt{int} to \texttt{short}
→ For memory-bound kernels, may improve perf. by factor of two
→ Should only be performed if correctness is guaranteed (always or special cases)

Other more complex solutions: in-place calculations
CAPACITY MISS TRAFFIC

- Calculation: pencil and paper or performance counters (e.g. L1 evictions due to replacement (because of capacity miss rather than L1 invalidation))
- Caches, local stores: finite capacity

Caches:
- Suppose, kernel’s working set exceeds cache capacity
  
  HW detects that data must be swapped out → capacity miss!
  
  → increase in DRAM memory traffic
  
  → reduced AI

  → if perf. limited by mem BW, it diminishes by proportional amount

Memory:
- Suppose, kernel’s working set exceeds Memory size
CAPACITY MISS TRAFFIC

- Calculation: pencil and paper or performance counters
  (e.g. L1 evictions due to replacement (because of capacity miss rather than L1 invalidation))
- Caches, local stores: finite capacity

Caches:
- Suppose, kernel’s working set exceeds cache capacity
  HW detects that data must be swapped out → capacity miss!
    → increase in DRAM memory traffic
    → reduced AI
    → if perf. limited by mem BW, it diminishes by proportional amount

Memory:
- Suppose, kernel’s working set exceeds Memory size
  → program will not function correctly!
Question:

- Most common solution to eliminate capacity misses for cache-based architectures?
Question:
• Most common solution to eliminate capacity misses for cache-based architectures?

• Cache blocking
  → restructure loops to reduce working set size
  → and maximize AI

Example:
```c
for ( i=0; i<N; i++) {...}
for ( i=0; i<N; i+=B)
  for ( j=i; j<min(N,B); j++) {...}
```
WRITE ALLOCATION TRAFFIC

• Most caches write-allocate policy
  (upon write-miss, cache evicts selected line, load target line from main memory)

→ Result: write generates 2x memory traffic as reads
  (cache line fill + a write back; versus one fill)

• Often wasteful on scientific code where large blocks of arrays are immediately written without being read!

• No benefit in having loaded the cache line when next memory operations will erase existing data
  → write-fill was unnecessary → hence should be denoted as AI wall

Solution:
• Modern arch. use SSE’s cache bypass instruction movntpd
  → bypasses cache all together + write to write combining buffers
  → eliminates write-allocation traffic + reduced cache pressure
CONFLICT MISS TRAFFIC

- Calculation: must use performance counters (see PAPI lecture)

- Caches are not fully associative (unlike local stores)
  (hard to build fast + fully associative caches as there is large bookkeeping overhead)
  → i.e. depending on address, only certain locations in cache used to store requested cache line (a set)

**Example:**

- Which memory locations can be cached by which cache locations:
CONFLICT MISS TRAFFIC

- If associativity of a set is exhausted, then one element from this set must be evicted
  → Result: Conflict Miss + redundant memory traffic 😞
- Happens often on $2^N$ problem sizes because it’s a multiple of the number of sets in a cache
- difficult to track down due to complexities of certain memory access patterns

Solution:
- **Array Padding**
  → for many well-structured codes
  → adds additional, unused space between array or dims of array
  → ensures that different sets are accessed + conflict misses avoided

**Example**: 1D array padding:

```plaintext
float A[2][4096];  →  float A[2][4096+64];
```
Roofline Model:

- a performance model intended to provide performance intuition for kernel analysis and optimization
- Upper bound to performance
- Tighter performance bounds may be refined through use of ceilings
  - Bandwidth ceilings
  - In-core ceilings
  - AI walls

Reference:


*Performance Tuning of Scientific Applications*