Vectorization

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

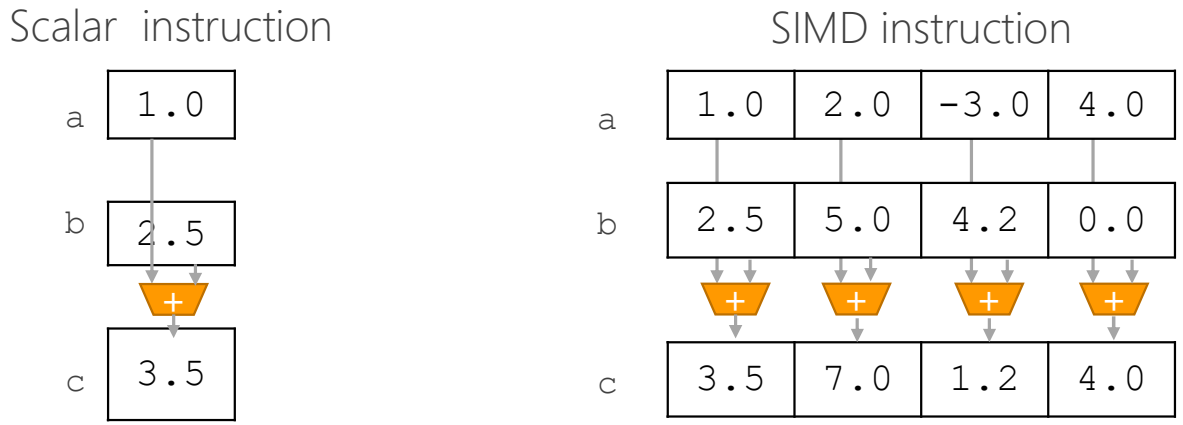
Outline

- Vector ISA
- AVX Intrinsic
  - registers
  - load and memory alignment
  - addition and subtraction
  - multiplication and division
  - FMA

SIMD Instructions

- Vector instructions
  - SIMD: Single Instruction – Multiple Data
  - special instructions that work with arrays (pack) of elements
    - typically: short, length fixed by the hw, not necessarily fixed by the ISA

- Example: `c = a + b`

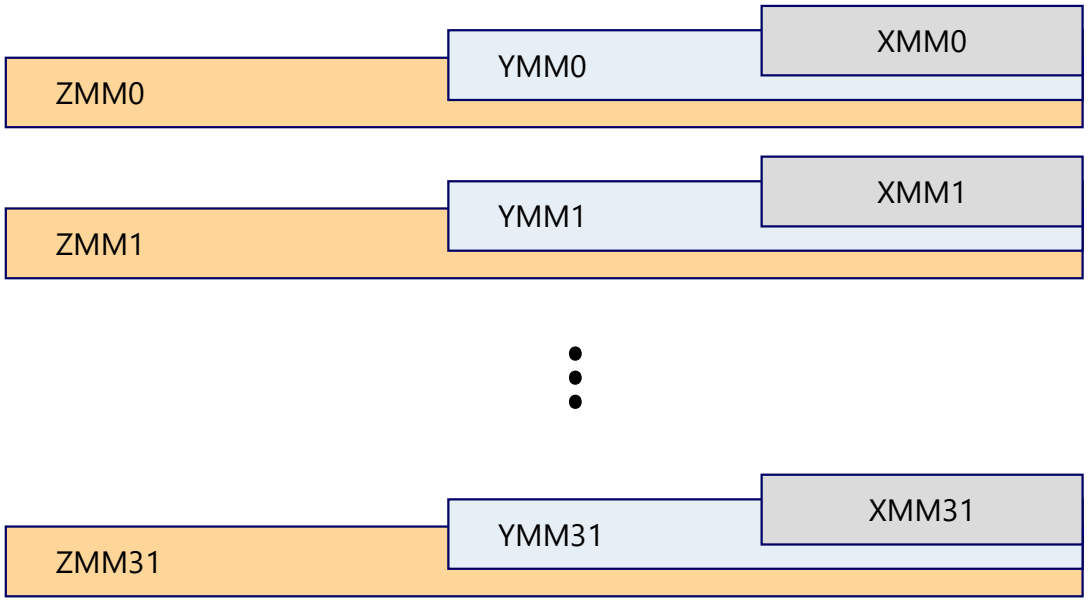


SIMD Extensions

ISA	SIMD Extension	Year
PA-RISC	Multimedia Acceleration eXtensions (MAX)	1994
SPARC	Visual Instruction Set (VIS)	1995
Alpha	Motion Video Instructions (MVI)	1996
MIPS	Multimedia Acceleration Instructions (MDMX)	1996
x86	MultiMedia eXtensions (MMX)	1996
PowerPC	Altivec	1998
X86	Streaming SIMD Extensions (SSE, SSE2, SSE3, SSE4)	1999-2007
ARM	Advanced SIMD Extension (NEON)	2005
X86	Advanced Vector eXtensions (AVX, AVX2, AVX-512)	2011-2013-2017
ARM	Scalable Vector Extension (SVE, SVE2)	2020-2021

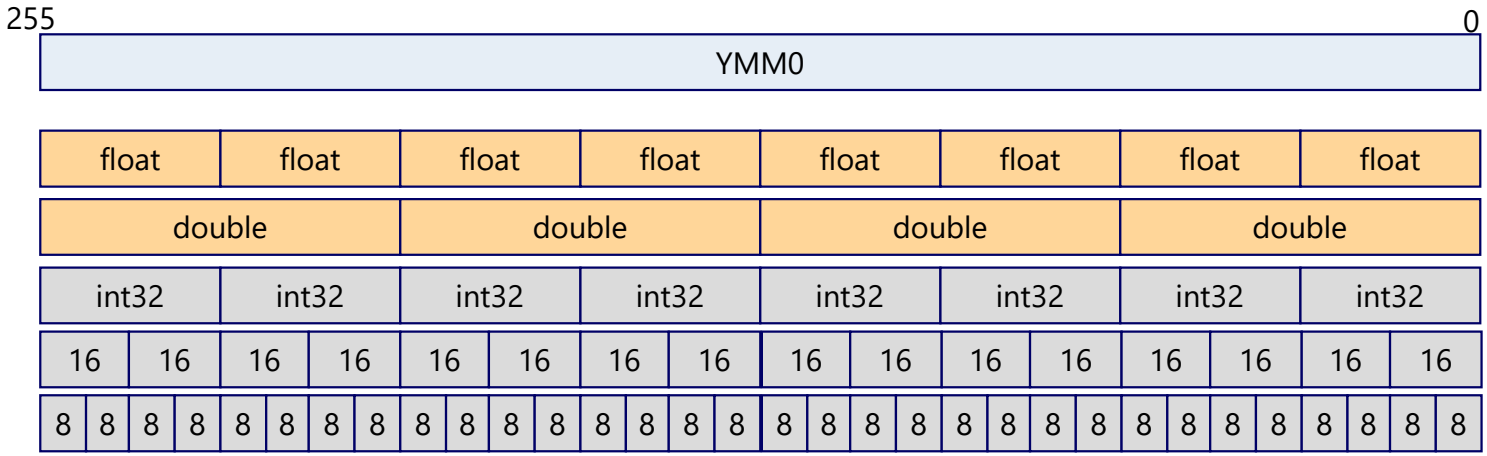
AVX/AVX2 Registers and data type

- Vector registers of different size
- XMM0 – XMM15
  - 128-bit registers
  - SSE
- YMM0 – YMM15
  - 256-bit registers
  - AVX, AVX2
- ZMM0 – ZMM31
  - 512-bit registers
  - AVX-512



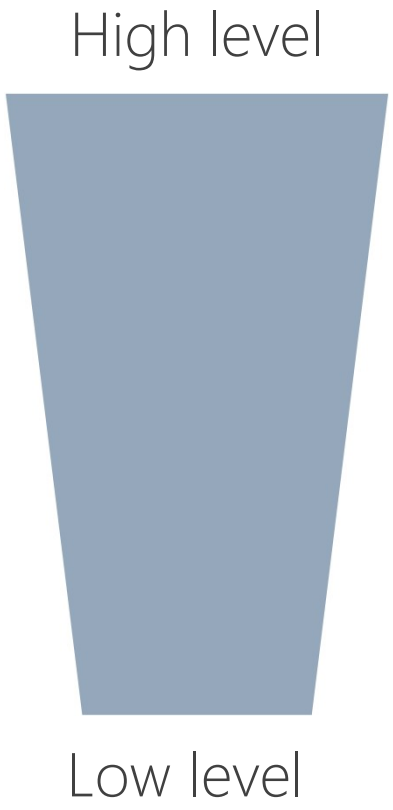
AVX/AVX2 Registers and Data Type

- Vector registers can be mapped on different data types
  - a YMM AVX vectors can contain up to 256 bits of data, e.g.
    - four doubles (4 x 64 bits = 256)
    - eight floats (8 x 32 bits = 256)
    - eight ints (8 x 32 bits = 256)



Programming Model for Vector

- Automatic vectorization
  - all made automatically by the compiler
- SIMD-optimized libraries
  - E.g., FFTW, GROMACS, MKL ,…
- Data parallel programming models with SIMD support
  - E.g., OpenCL, SYCL, ispc, TBB, OpenMP SIMD pragma
  - Implicit use of vector
- SIMD intrinsics
  - express vector manually
  - leave register allocation and instr. scheduling to the compiler
- Assembly
  - all made manually by the developer

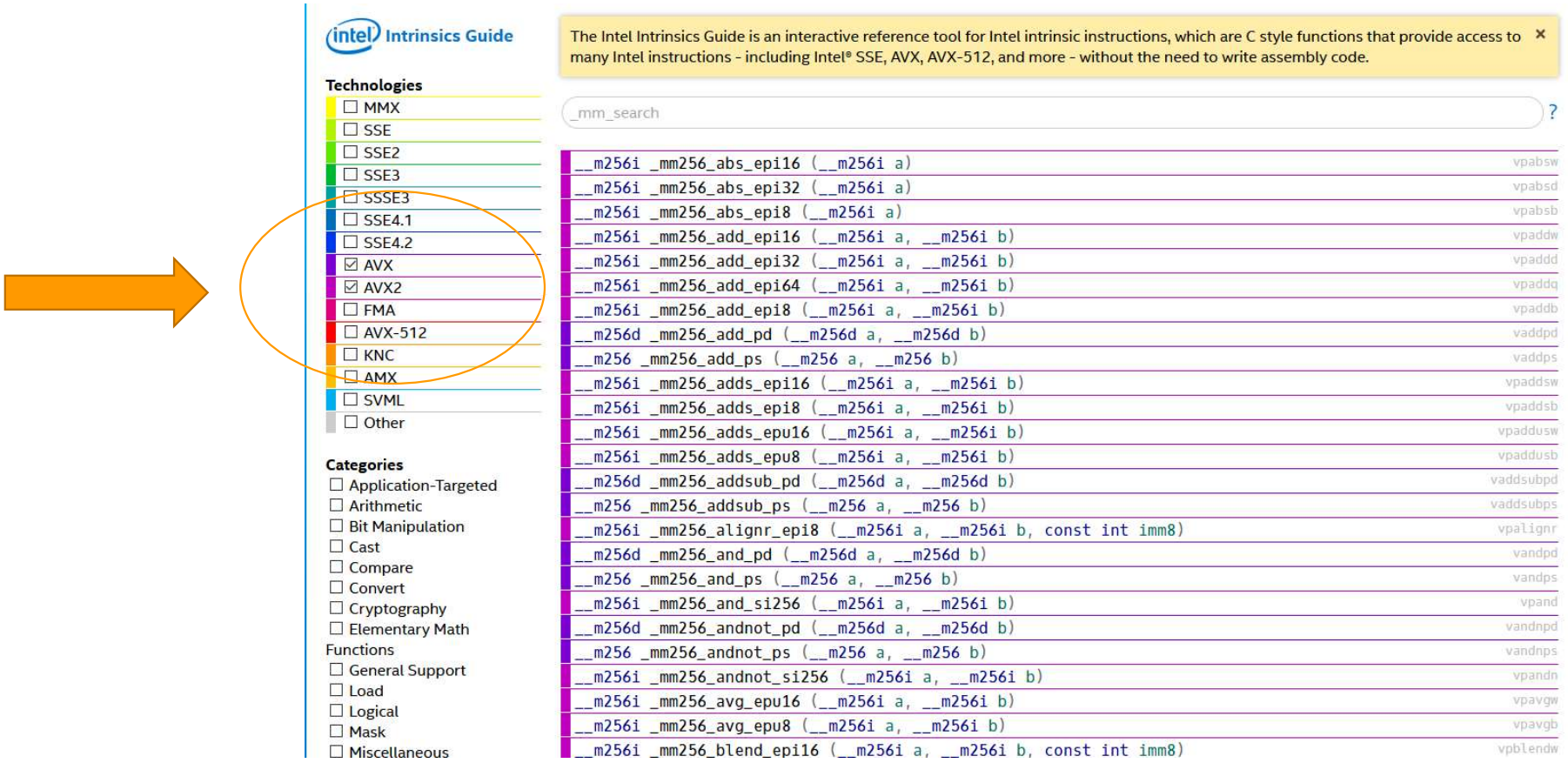


SIMD Intrinsics Programming

- Low-level C functions for SIMD programming
  - an instrinsics function is mapped on one (sometime more than one) vectorial instruction
- Example:
  - the intrinsic function `_mm256_add_ps()` maps directly to `vaddps`
- Combining the performance of assembly with the convenience of a high-level function
  - register allocation and instruction scheduling performed by the compiler

Intel Intrinsics Guide

https://software.intel.com/sites/landingpage/IntrinsicsGuide



Intel Intrinsics Guide

https://software.intel.com/sites/landingpage/IntrinsicsGuide

```
__m256 _mm256_add_ps (__m256 a, __m256 b)
```

vaddps

Synopsis

```
__m256 _mm256_add_ps (__m256 a, __m256 b)
#include <immintrin.h>
Instruction: vaddps ymm, ymm, ymm
CPUID Flags: AVX
```

Description

Add packed single-precision (32-bit) floating-point elements in a and b, and store the results in dst.

Operation

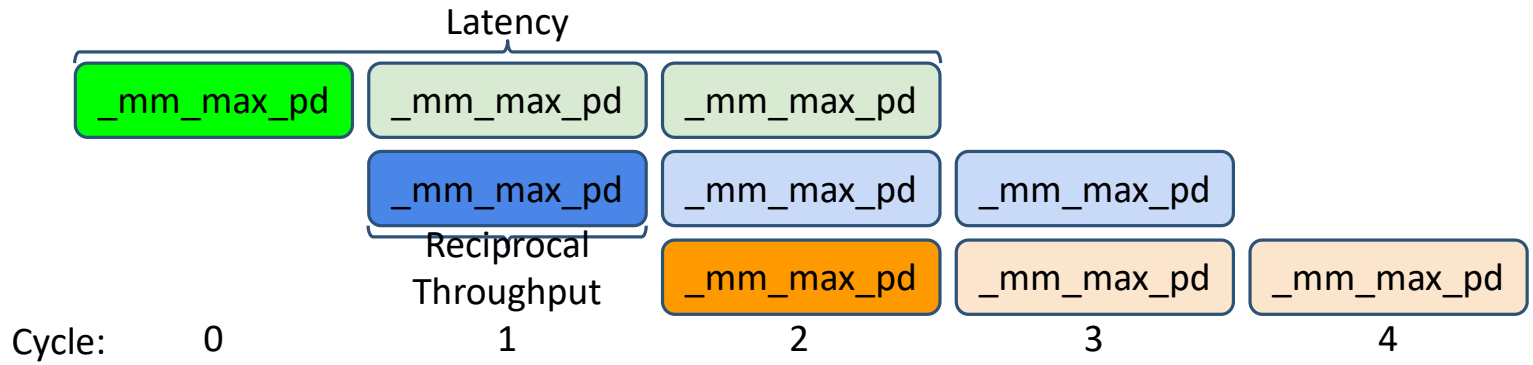
```
FOR j := 0 to 7
        i := j*32
        dst[i+31:i] := a[i+31:i] + b[i+31:i]
ENDFOR
dst[MAX:256] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	4	0.5
Skylake	4	0.5
Broadwell	3	1
Haswell	3	1
Ivy Bridge	3	1

Latency and Throughput

- Latency = How many cycles it takes before the next dependent operation can start
- Throughput = How many independent operations can run per cycle
- Example: _mm_max_pd



Latency and Throughput

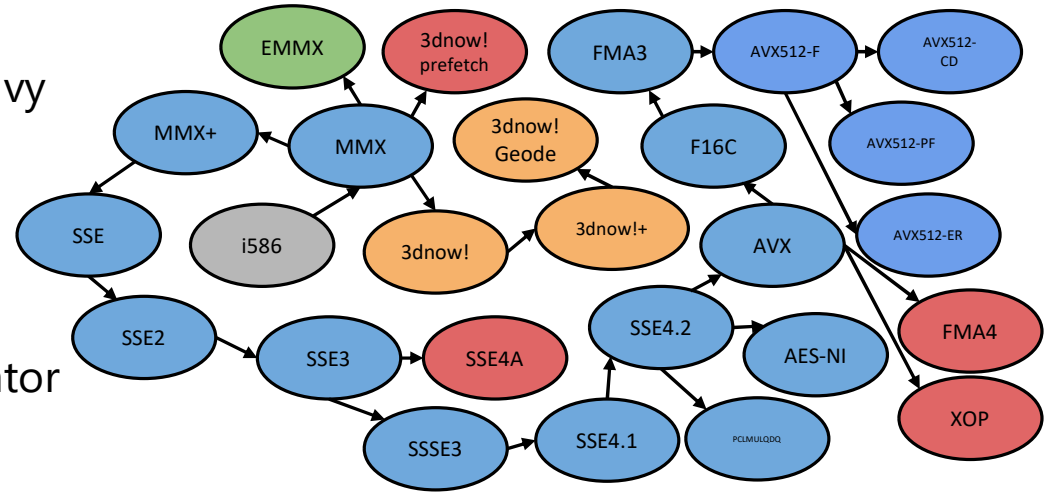
Latency cost >> Throughput cost

- Performance of independent operations is bounded by instructions throughput
- Performance of dependent operations is bounded by instructions latency

- To get better performance increase parallelism of your code
  - create opportunities to run instructions in parallel
- Loop unrolling often helps to break dependencies

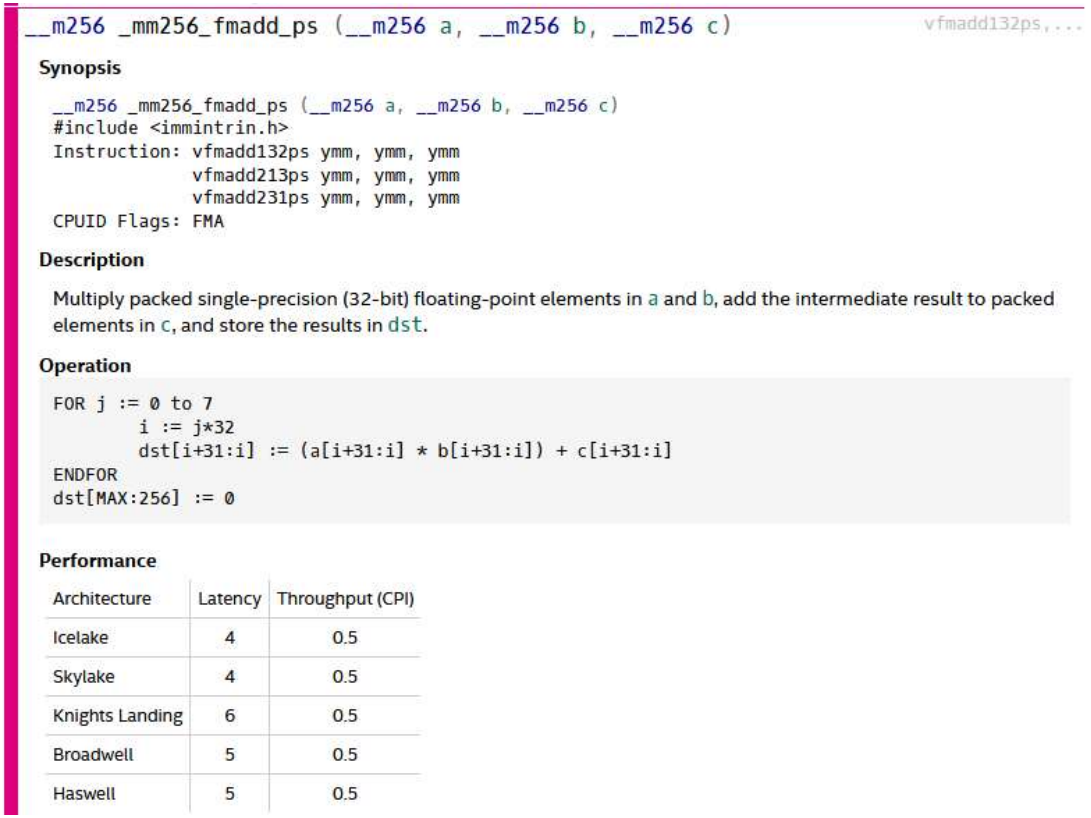
Intel AVX/AVX2

- AVX
  - Intel's Sandy Bridge/Sandy Bridge E/Ivy Bridge/Ivy Bridge E
  - Intel's Haswell/Haswell E/Broadwell/Broadwell E
  - AMD's Bulldozer/Piledriver/Steamroller/Excavator
- AVX2
  - Intel's Haswell/Haswell E/Broadwell/Broadwell E
  - AMD's Excavator



Intrinsics to Instruction Mapping

- intrinsics `_mm256_fmadd_ps` maps to the following instructions
  1. `vfmadd132ps`
  2. `vfmadd213ps`
  3. `vfmadd231ps`
- Despite this, it is very fast!



`__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c)` vfmadd132ps,...

Synopsis

```
__m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c)
#include <immintrin.h>
Instruction: vfmadd132ps ymm, ymm, ymm
             vfmadd213ps ymm, ymm, ymm
             vfmadd231ps ymm, ymm, ymm
CPUID Flags: FMA
```

Description

Multiply packed single-precision (32-bit) floating-point elements in a and b, add the intermediate result to packed elements in c, and store the results in dst.

Operation

```
FOR j := 0 to 7
        i := j*32
        dst[i+31:i] := (a[i+31:i] * b[i+31:i]) + c[i+31:i]
ENDFOR
dst[MAX:256] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	4	0.5
Skylake	4	0.5
Knights Landing	6	0.5
Broadwell	5	0.5
Haswell	5	0.5

AVX/AVX2 Data Types

`__m128`	128-bit vector containing 4 floats
`__m128d`	128-bit vector containing 2 doubles
`__m128i`	128-bit vector containing integers
`__m256`	256-bit vector containing 8 floats
`__m256d`	256-bit vector containing 4 doubles
`__m256i`	256-bit vector containing integers

- each type starts with two underscores, an `m`, and the width of the vector in bits
  - AVX/AVX2 -> 256 bits, AVX512 -> 512 bits
- If a vector type ends in `d`, it contains doubles, and if it doesn't have a suffix, it contains floats
- `__m128i` and `__m256i` vectors integer vector types e.g., 32 chars, 16 shorts, 8 ints, or 4 longs., can be signed or unsigned

AVX/AVX2 Function Naming Conventions

`_mm<bit_width>_<name>_<data_type>`

- `<bit_width>` identifies the size of the vector returned by the function. For 128-bit vectors, this is empty. For 256-bit vectors, this is set to 256.
  - `<name>` describes the operation performed by the intrinsic
  - `<data_type>` identifies the data type of the function's primary arguments
- `<data_type>` identifies the content of the input values:
  - `ps` - vectors contain floats (packed single-precision)
  - `pd` - vectors contain doubles (packed double-precision)
  - `epi8/epi16/epi32/epi64` - vectors contain 8-bit/16-bit/32-bit/64-bit signed integers
  - `epu8/epu16/epu32/epu64` - vectors contain 8-bit/16-bit/32-bit/64-bit unsigned integers
  - `si128/si256` - unspecified 128-bit vector or 256-bit vector
  - `m128/m128i/m128d/m256/m256i/m256d` - identifies input vector types when they're different than the type of the returned vector

AVX Example: Vector Difference

```
#include <immintrin.h>
#include <stdio.h>

int main() {
  /* Initialize the two argument vectors */
  __m256 evens = _mm256_set_ps(2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0);
  __m256 odds  = _mm256_set_ps(1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0);

  /* Compute the difference between the two vectors */
  __m256 result = _mm256_sub_ps(evens, odds);

  /* Display the elements of the result vector */
  float* f = (float*)&result;
  printf("%f %f %f %f %f %f %f %f\n",
    f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7]);
  return 0;
}
```

```
gcc -mavx -o hello_avx hello_avx.c
```

Initialization Intrinsics with Scalar Values

`_mm256_setzero_ps/pd`	Returns a floating-point vector filled with zero
`_mm256_setzero_si256`	Returns an integer vector whose bytes are set to zero
`_mm256_set1_ps/pd`	Fill a vector with a floating-point value
`_mm256_set1_epi8/epi16/epi32/epi64`	Fill a vector with an integer
`_mm256_set_ps/pd`	Initialize a vector with eight floats (`ps`) or four doubles (`pd`)
`_mm256_set_m128/m128d/m128i`	Initialize a 256-bit vector with two 128-bit vectors
`_mm256_setr_ps/pd`	Initialize a vector with eight floats (`ps`) or four doubles (`pd`) in reverse order
`_mm256_setr_epi8/epi16/epi32/epi64`	Initialize a vector with integers in reverse order

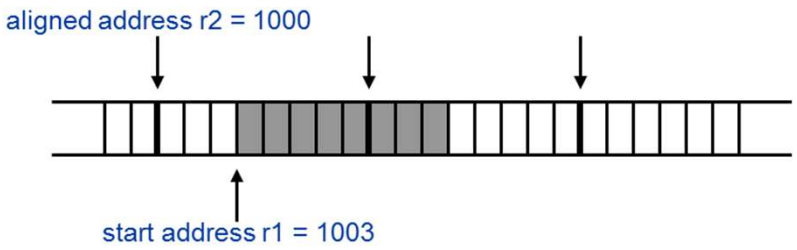
Loading from Memory

`_mm256_load_ps/pd`	Loads a floating-point vector from an aligned memory address
`_mm256_load_si256`	Loads an integer vector from an aligned memory address
`_mm256_loadu_ps/pd`	Loads a floating-point vector from an unaligned memory address
`_mm256_loadu_si256`	Loads an integer vector from an unaligned memory address
`_mm_maskload_ps/pd` `_mm256_maskload_ps/pd`	Load portions of a 128-bit/256-bit floating-point vector according to a mask
`_mm_maskload_epi32/64` `_mm256_maskload_epi32/64`	Load portions of a 128-bit/256-bit integer vector according to a mask

- Different kind of load instructions
  - aligned load
  - unaligned load
  - masked load
  - also: gather functions that load indexed data from memory

Memory Alignment

- Data must be aligned at an N-byte boundary
  - 64-bit (MMX):
  - 128-bit (SSE, Altivec)
  - 256-bit (AVX)
- Unaligned accesses
  - Load two aligned positions
  - Shift
  - Merge
- Handling alignment
  - Software: the programmer has to do it
  - OS exception: very slow
  - Hardware: transparent to the programmer



Memory Alignment

- When loading data into vectors, memory alignment is important
  - each `_mm256_load_*` intrinsic accepts a memory address that must be aligned on a 32-byte boundary, i.e., the address must be divisible by 32
  - any attempt to load unaligned data with `_m256_load_*` produces a segmentation fault.
- Example

```
float* aligned_floats = (float*)aligned_alloc(32, 64 * sizeof(float));
... Initialize data ...
__m256 vec = _mm256_load_ps(aligned_floats);
```

- If the data isn't aligned at a 32-bit boundary, the `_m256_loadu_*` functions should be used instead
- Example

```
float* unaligned_floats = (float*)malloc(64 * sizeof(float));
... Initialize data ...
__m256 vec = _mm256_loadu_ps(unaligned_floats);
```

Masked Load

- Suppose you want to process a float array of 11 float in AVX
  - 11 isn't divisible by 8, the last five floats of the second `__m256` vector need to be set to zero so they don't affect the computation
- This selective loading can be accomplished with the `_maskload_` functions
  - each `_maskload_` function accepts two arguments:
    - a memory address and an integer vector with the same number of elements as the returned vector
    - for each element in the integer vector whose highest bit is one, the corresponding element in the returned vector is read from memory
    - if the highest bit in the integer vector is zero, the corresponding element in the returned vector is set to zero
- Related instruction: `_blendv_`

Masked Load Example

```
int array[8]   = { 1, 2, 3, 4, 5, 6, 7, 8 };
__m256i mask   = _mm256_setr_epi32(-2, -42, 10, 0, -100, 0, 4, 8);
__m256i result = _mm256_maskload_epi32(array, mask);

int* res = (int*)&result;
printf("%d %d %d %d %d %d %d %d\n",
  res[0], res[1], res[2], res[3], res[4], res[5], res[6], res[7]);
```

- Output:

```
1 2 0 0 5 0 0 0
```

- Note: `setr` loads the mask in reversed order

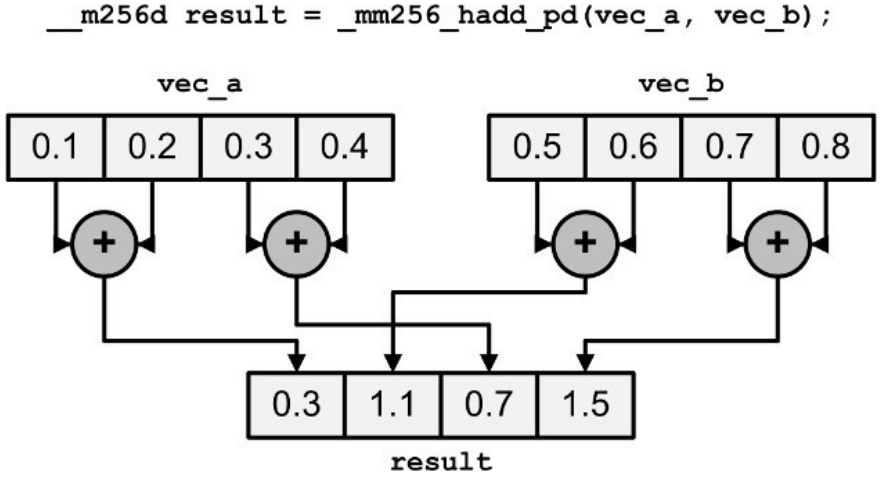
Arithmetic Intrinsics

`_mm256_add_ps/pd`	Add two floating-point vectors
`_mm256_sub_ps/pd`	Subtract two floating-point vectors
`_mm256_add_epi8/16/32/64`	Add two integer vectors
`_mm236_sub_epi8/16/32/64`	Subtract two integer vectors
`_mm256_adds_epi8/16` `_mm256_adds_epu8/16`	Add two integer vectors with saturation
`_mm256_subs_epi8/16` `_mm256_subs_epu8/16`	Subtract two integer vectors with saturation

- Saturated arithmetic: `_adds_` / `_subs_`
  - when the result requires more memory than the vector can store
  - clamp the result to the minimum/maximum value that can be stored

Horizontal Addition and Subtraction

- `_hadd_` /`_hsub_` functions perform addition and subtraction horizontally
  - add or subtract adjacent elements within each vector
  - results are stored in an interleaved fashion
- Example: `_mm256_hadd_pd` horizontally adds double vectors a and b

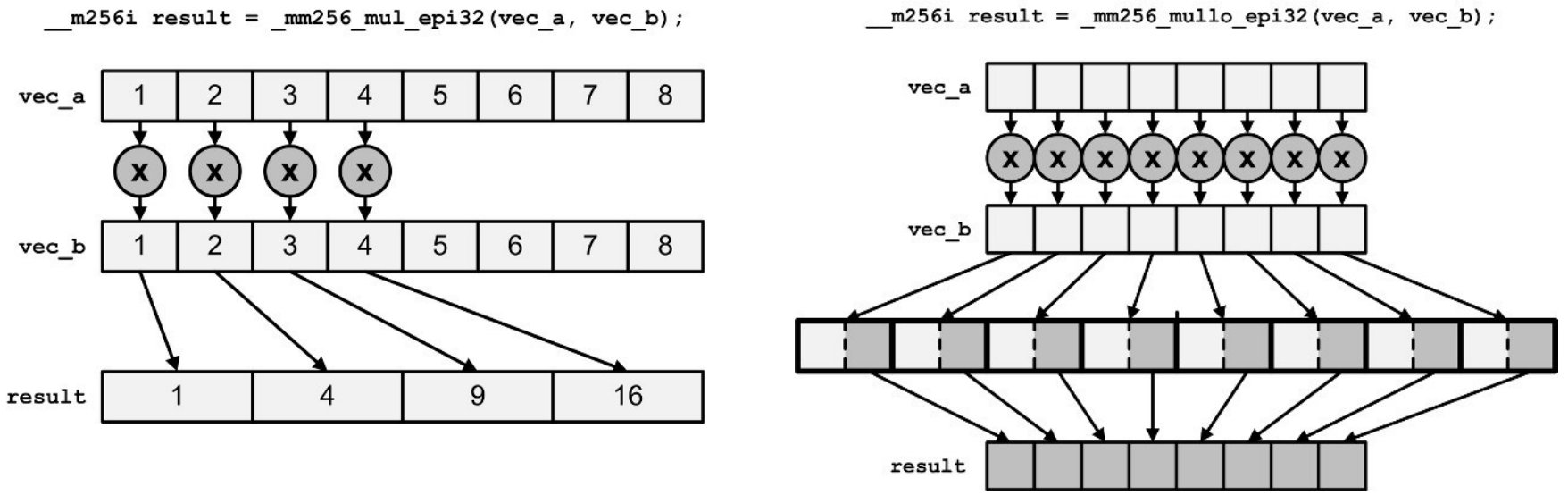


Multiplication and Division

`_mm256_add_ps/pd`	Multiply two floating-point vectors
`_mm256_mul_epi32/epu32`	Multiply the lowest four elements of vectors containing 32-bit integers
`_mm256_mullo_epi16/32`	Multiply integers and store low halves
`_mm256_mulhi_epi16/epu16`	Multiply integers and store high halves
`_mm256_mulhrs_epi16`	Multiply 16-bit elements to form 32-bit elements
`_mm256_div_ps/pd`	Divide two floating-point vectors

- If you multiply 2 N-bit numbers, the results can take up to 2N bits
- Low and high
  - `_mullo_` multiply every element but only store the low half of each product
  - `_mulhi_` multiply every element but only store the high half of each product

Low and High with Integer Multiplication



- Low and high can be combined depending on the required accuracy
- Note: in C/C++, integer overflow is
  - unsigned integer: modulo power of two
  - signed integer: undefined behavior

FMA

`_mm_fmadd_ps/pd/_mm256_fmadd_ps/pd`	Multiply two vectors and add the product to a third (res = a * b + c)
`_mm_fmsub_ps/pd/_mm256_fmsub_ps/pd`	Multiply two vectors and subtract a vector from the product (res = a * b - c)

- FMA: fuse multiplication and addition together
  - the result of multiplying two N-bit numbers can occupy 2N bits
  - when you multiply two floating-point values, a and b, the result is really `round(a*b)`, where `round(x)` returns the floating-point value closest to `x`
  - with FMA: instead of `round(round(a*b)+c)`, they return `round(a*b+c)`
  - as a result, these instructions provide greater speed and accuracy than performing multiplication and addition separately

FMA Example

- Example without FMA vectors

```
void multiply_and_add(const float*a, const float*b, const float*c, float*d) {
  for(int i=0; i<8; i++) {
    d[i] = a[i] * b[i];
    d[i] = d[i] + c[i];
  }
}
```

- Example with FMA

```
__m256 multiply_and_add(__m256 a, __m256 b, __m256 c) {

  return _mm256_fmadd_ps(a, b, c);
}
```

AVX Lab Exercises

1. Implement an AVX program that adds two arrays of floats
2. Implement an AVX program that adds the only odd (or even) elements of two arrays of floats
3. Implement a dot product
4. Implement the following loop

```
uint8_t img_in1[SIZE], img_in2[SIZE], img_out[SIZE];
…
for (i=0; i<SIZE; i++){
   img_out[i] = img_in1[i];
   if (img_in[1] > 200)
      img_out[i] = img_in2[i];
}
```

- For all exercises:
  - input data must be dynamically allocated
  - input size can be a multiple of 8
  - compare results against a simple loop in `-O0`, with and without data alignment

Data Reorganization

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

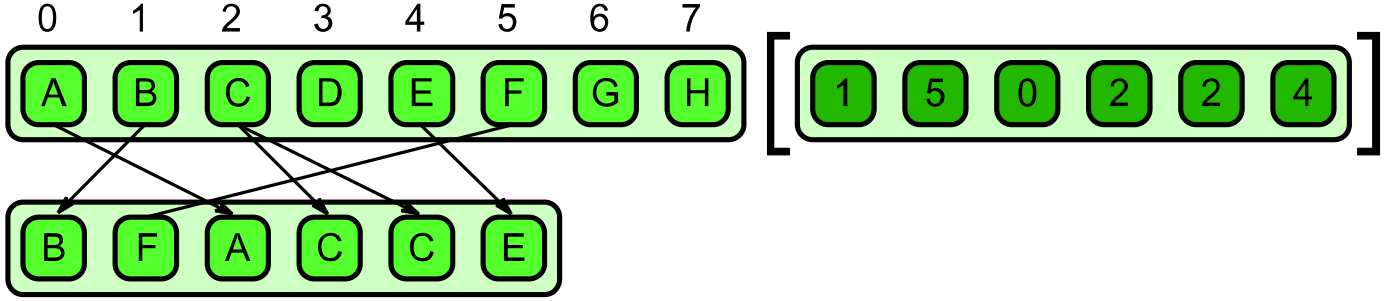
- Data reorganization patterns
  - Gather
  - Scatter
  - Pack
- AoS vs SoA
- Intel AVX
  - Permuting
  - Shuffling
  - AVX complex multiplication

Data Movement

- Performance is often more limited by data movement than by computation
  - Transferring data across memory layers is costly
    - locality is important to minimize data access times
    - data organization and layout can impact this
  - Transferring data across networks can take many cycles
    - attempting to minimize the # messages and overhead is important
- Data movement also costs more in power
  - for “data intensive” application, it is a good idea to design the data movement first
  - design the computation around the data movements
  - applications such as search and sorting are all about data movement and reorganization

Gather Pattern

- Gather pattern creates a (output) collection of data by reading from another (input) data collection
  - given a collection of indices
  - read data from the input collection at each index
  - write data to the output collection in index order
- Transfers data from source collection to output collection
  - element type of output collection is the same as the source
  - size of the output collection is that of the index collection

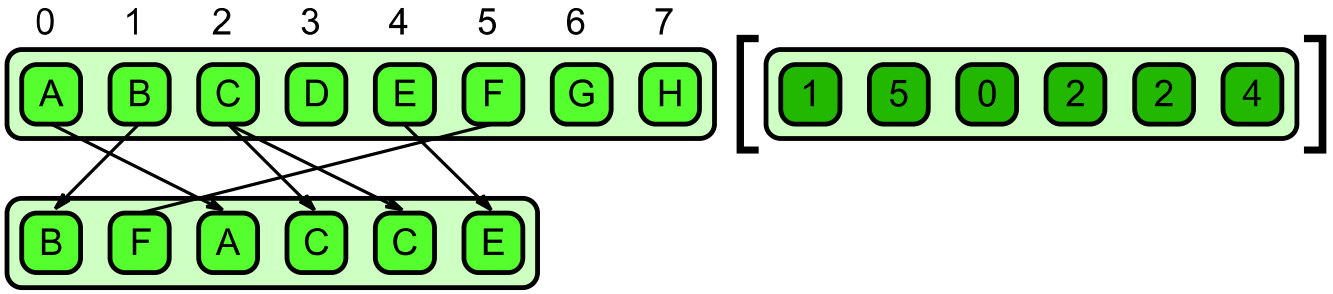


Gather Pattern

- Serial implementation in C

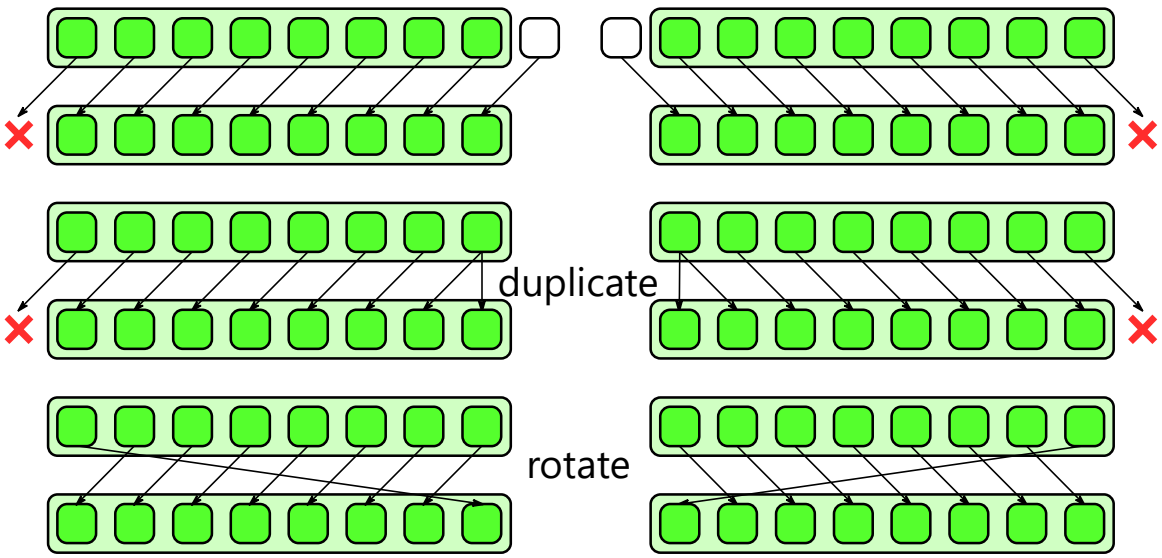
```
const int N = 8;
const int M = 6;
char A[] =  { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'}; // input
int id[M] = { 1, 5, 0, 2, 2, 4 };
char B[M]; // output

for (int i = 0; i < M; i++) {
  int j = id[i];
  B[i] = A[j];
}
```



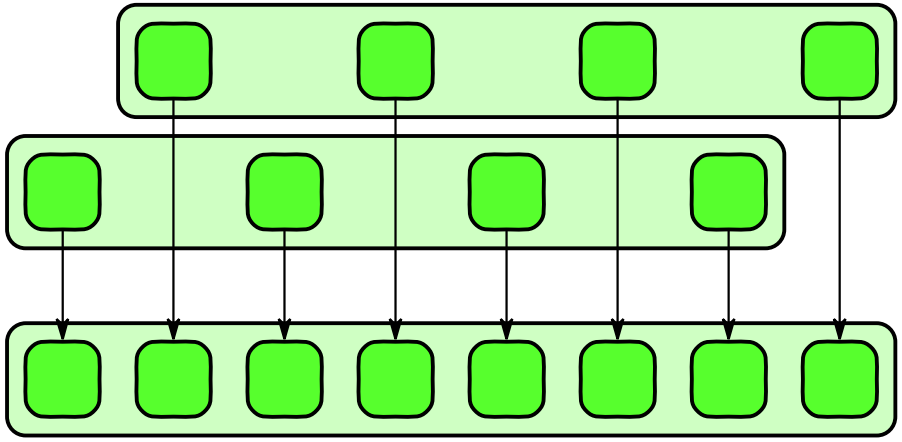
Gather Special Case: Shifts

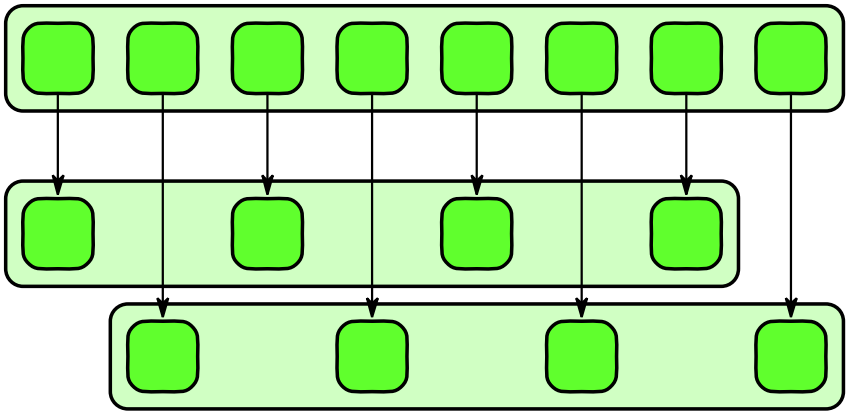
- Shifts
  - moves data to the left or right in memory
  - data accesses are offset by fixed distances
  - variants from how boundary conditions are handled
  - supported by vector instructions



Gather Special Case: Zip and Unzip

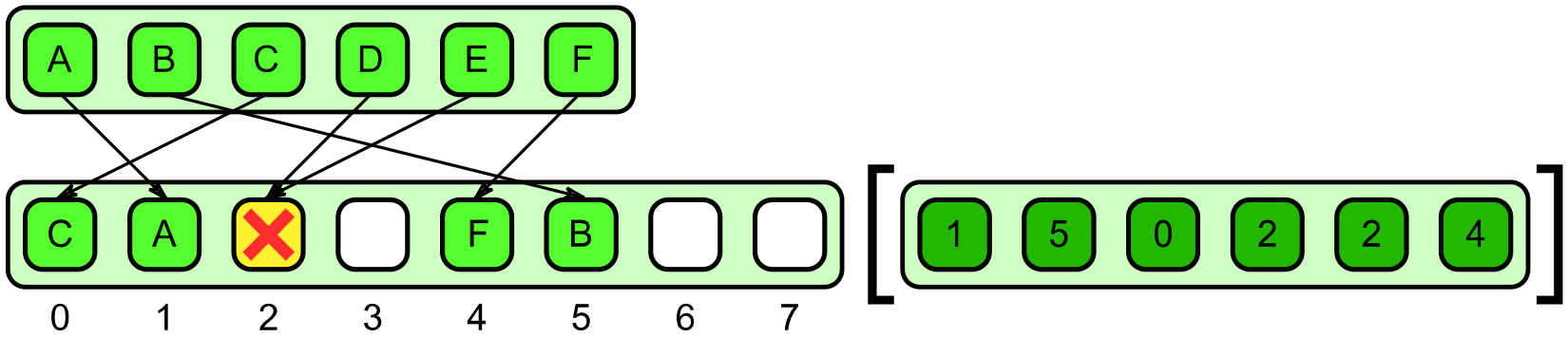
- Zip: Function is to interleaves data
  - like a zipper
- Unzip: Reverses a zip
  - extracts sub-arrays at certain offsets and strides from an input array





Pattern: Scatter

- Given a collection of input data and a collection of write locations, scatter data to the output collection
- Writes to the same location are possible
  - parallel writes to the same location are collisions

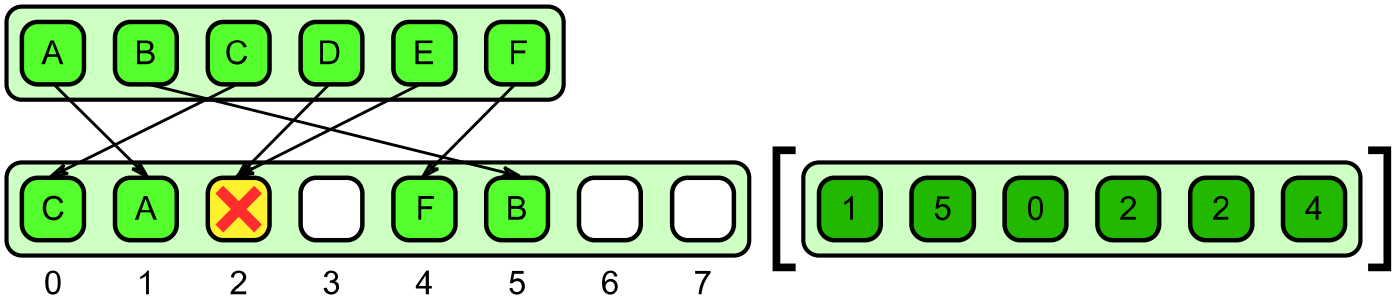


Scatter Pattern

- Serial implementation in C

```
const int N = 6;
const int M = 8;
char A[N] = { 'A', 'B', 'C', 'D', 'E', 'F'}; // input
int id[N] = { 1, 5, 0, 2, 2, 4 };
char B[M]; // output

for (int i = 0; i < N; i++) {
    int j = id[i];
    B[j] = A[i];
}
```

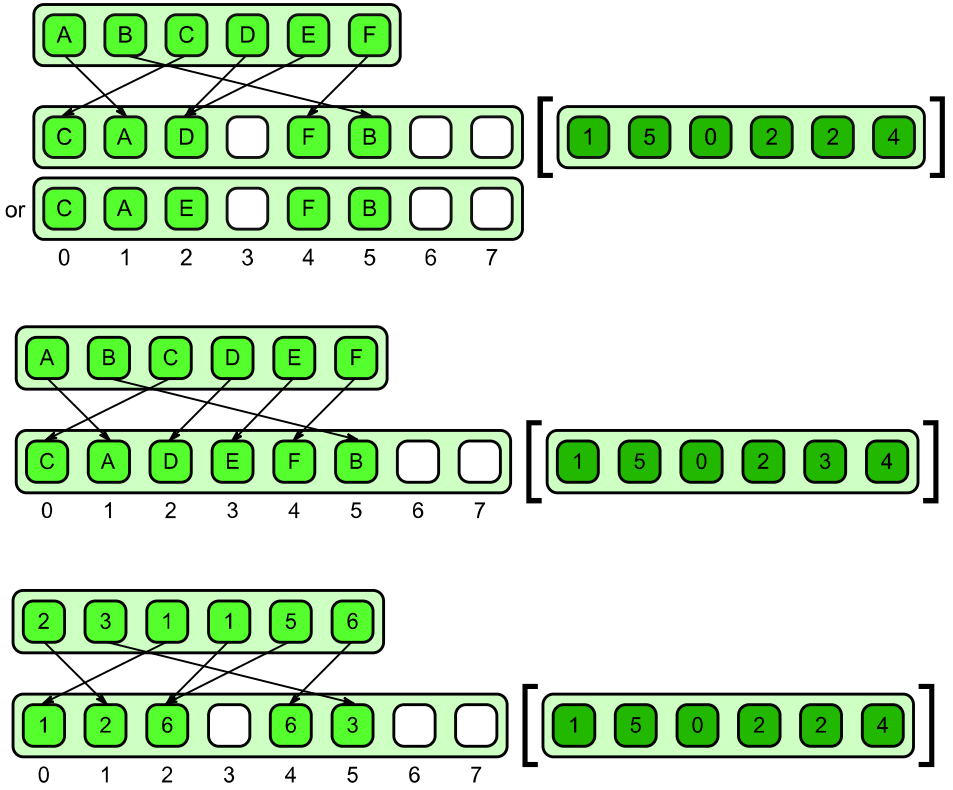


Scatter: Race Condition

- Race condition
  - two (or more) values being written to the same location in output collection
  - result is undefined unless enforce rules
  - need rules to resolve collisions

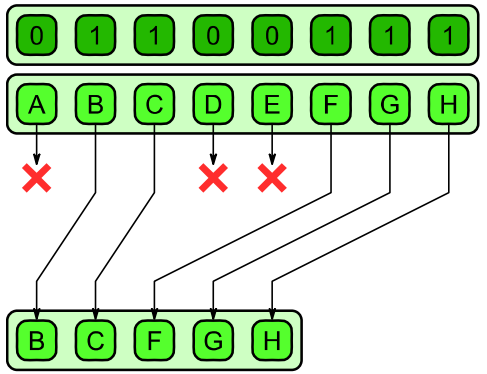
Scatter: Collision Resolution

- Atomic Scatter
  - no rule; non-deterministic approach
  - upon collision, one and only one of the values written to a location will be written in its entirety
  - e.g., either D and E at index 2
- Permutation Scatter
  - pattern simply states that collisions are illegal
  - output is a permutation of the input
  - e.g., FFT scrambling, matrix/image transpose, unpacking
- Merge Scatter
  - associative and commutative operators are provided to merge elements in case of a collision
    - required since scatters to a particular location could occur in any order
  - e.g., use addition as the merge operator

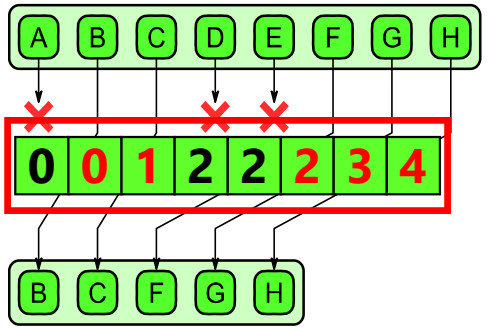


Pattern: Pack Algorithm

- Pack algorithm
  - used to eliminate unused elements from a collection
  - retained elements are moved so they are contiguous in memory

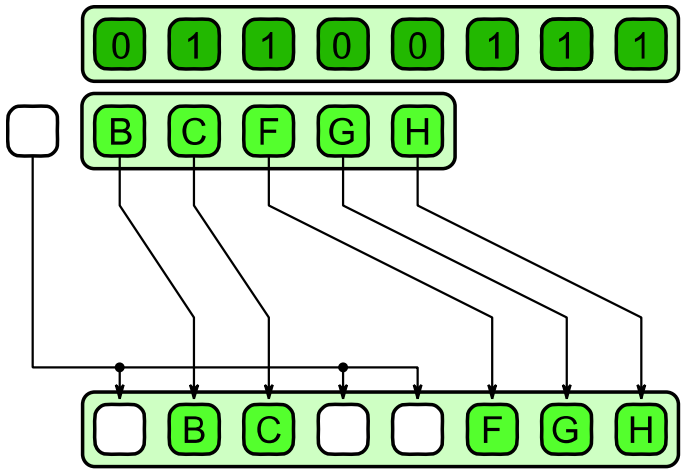


- Typical implementation steps
  1. Convert input booleans into integer 0's and 1's
  2. Exclusive scan of the array with addition operation
  3. Write value to output arrays based on offsets



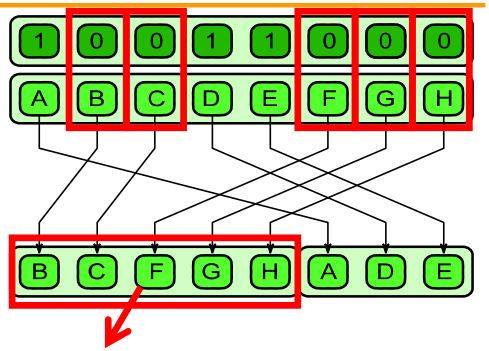
Pattern: Unpack

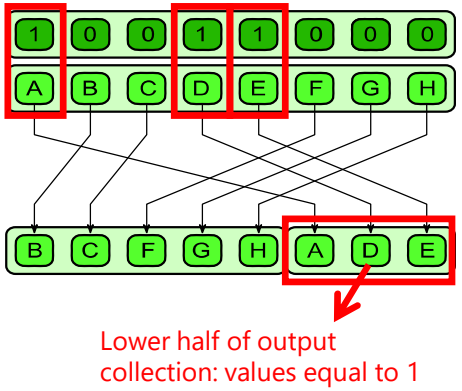
- Unpack algorithm
  - inverse of pack operation
  - given the same data on which elements were kept and which were discarded, spread elements back in their original locations



Pack Generalizations

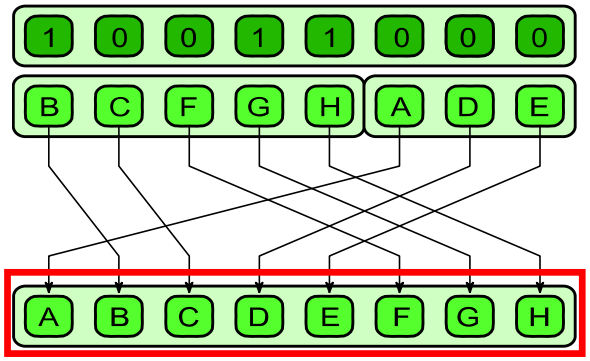
- Split
  - generalization of pack pattern
  - elements are moved to upper or lower half of output collection based on some state
  - does not lose information like pack



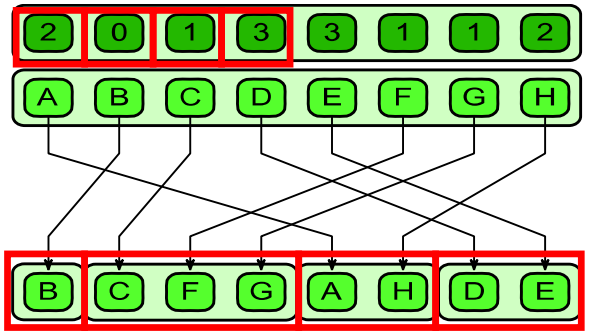


Pack Generalizations

- Unsplit
  - inverse of split
  - creates output collection based on original input collection



- Bin
  - generalized split to support more categories (>2)
  - examples
    - radix sort
    - pattern classification

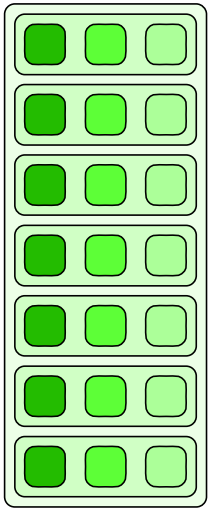


4 different categories = 4 bins

Array of Structures (AoS) vs Structure of Arrays (SoA)

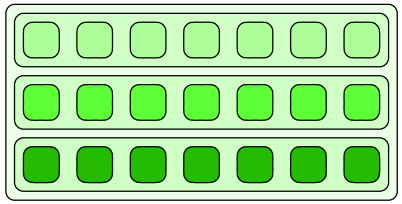
- AoS
  - most logical layout
  - difficult for (gather) reads and (scatter) writes; difficult for vectorization
  - may lead to better cache utilization if data is accessed randomly

```
struct AOS_Point {
  float x, y, z;
};
struct AOS_Point points[1024];
```



- SoA
  - separate each structure field
  - keep memory access contiguous if access over structure instances
  - often better for vectorization and avoidance of false sharing

```
struct SoA_point {
  float x[1024], y[1024], z[]1024;
};
struct SoA_point Points;
```



AoS vs SoA

- Code example

```
struct AOS_Point {
  float x, y, z;
};
struct AOS_Point points[1024];
```

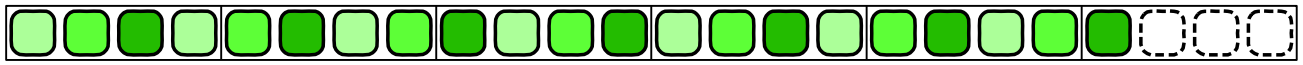
```
struct SoA_Point {
  float x[1024], y[1024], z[]1024;
};
struct SoA_Point points;
```

```
float dist[1024];
for(i=0; i<1024; i++){
  float x = points[i].x;
  float y = points[i].y;
  float z = points[i].z;
  float d = sqrtf(x*x + y*y + z*z);
  dist[i] = d;
}
```

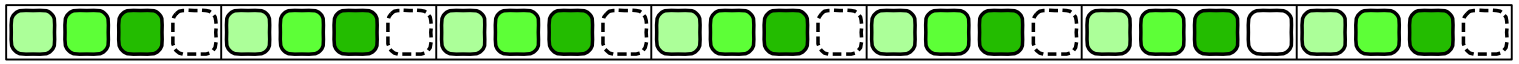
```
float dist[1024];
for(i=0; i<1024; i++){
  float x = points.x[i];
  float y = points.y[i];
  float z = points.z[i];
  float d = sqrtf(x*x + y*y + z*z);
  dist[i] = d;
}
```

Data Layout Options

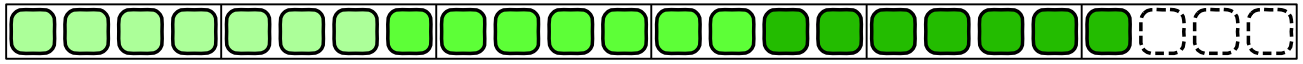
Array of Structures (AoS), padding at end



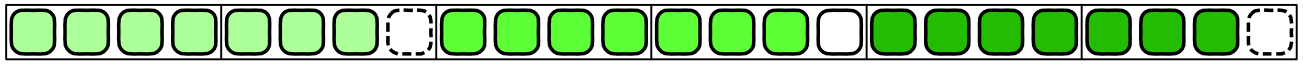
Array of Structures (AoS), padding after each structure



Structure of Arrays (SoA), padding at end



Structure of Arrays (SoA), padding after each component



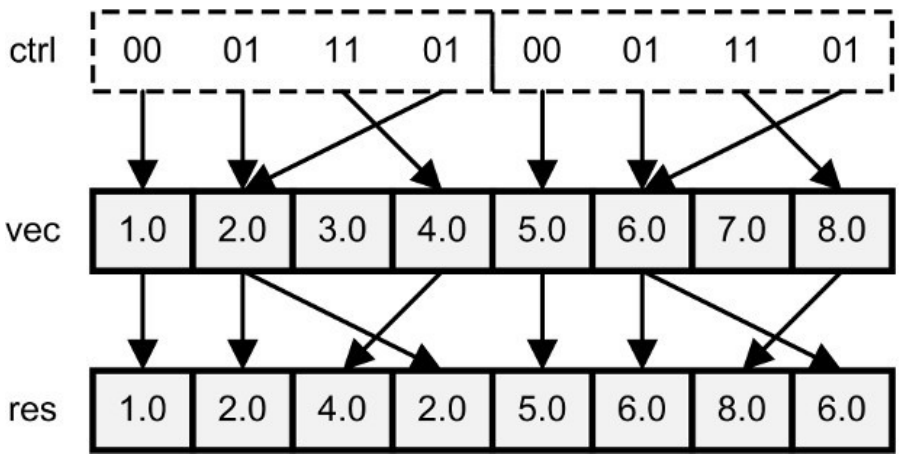
Reorganizing Data in AVX

- Many applications must rearrange vector elements to ensure that operations are performed properly
  - AVX/AVX2 provides a number of intrinsic functions for this purpose
- Two major categories
  - `_permute_` functions
  - `_shuffle_` functions

AVX Permuting

- `_permute_` intrinsic accepts two arguments
  - an input vector and an 8-bit control value
  - the bits of the control value determine which of the input vector's elements is inserted into the output
- Example
  - `_mm256_permute_ps`, each pair of control bits determines an upper and lower output element by selecting one of the upper or lower elements in the input vector

```
res = _mm256_permute_ps(vec, 0b01110100)
```



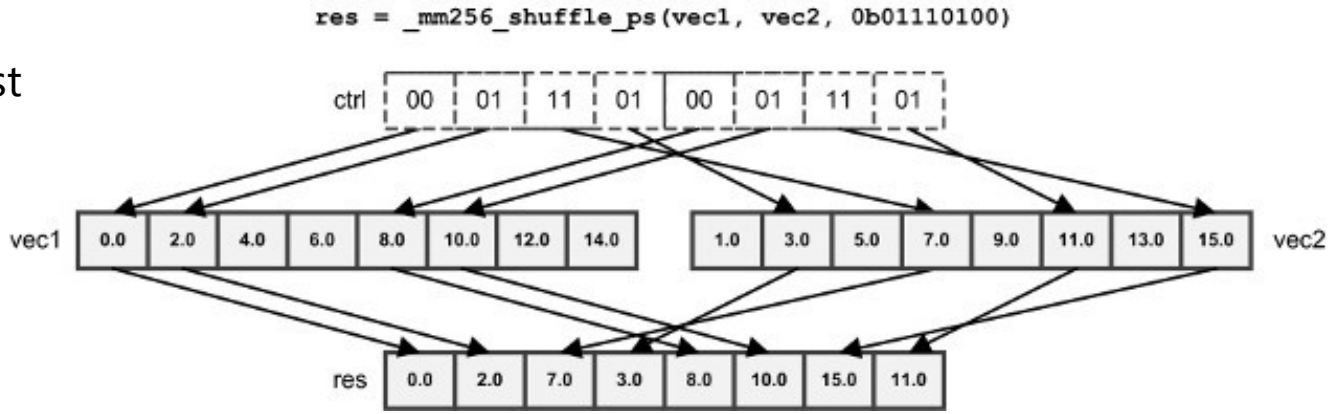
AVX Permuting

`_mm_permute_ps/pd/` `_mm256_permute_ps/pd`	Select elements from the input vector based on an 8-bit control value
`_mm256_permute4x64_pd/epi64`	Select 64-bit elements from the input vector based on an 8-bit control value
`_mm256_permute2f128_ps/pd/si256`	Select 128-bit chunks from two input vectors based on an 8-bit control value
`_mm_permutevar_ps/pd` `_mm256_permutevar_ps/pd`	Select elements from the input vector based on bits in an integer vector
`_mm256_permutevar8x32_ps/epi32`	Select 32-bit elements (`float`s and `int`s) using indices in an integer vector

- Note
  - `_mm256_permute_pd`, the low four bits of the control value select between adjacent pairs of doubles
  - `_mm256_permute4x4_pd` uses all of the control bits to select which 64-bit element is placed in the output
  - `_permute2f128_ intrinsics`, the control value selects 128-bit chunks from the two input vectors instead of selecting elements from one input vector
  - `_permutevar_` same operation as the `_permute_` intrinsics, but use vector of integer instead of instead of using 8-bit control values to select elements
  - the input vector of `_mm256_permute_ps` is a `_mm256`, so the integer vector is an _mm256i, the high bits of the integer vector perform selection in the same way as the bits of the 8-bit control values of the _permute_ intrinsics

AVX Shuffling

- `_shuffle_` intrinsics select elements from one or two input vectors and place them in the output vector
- Example: `_mm256_shuffle_ps`
  - only the high four bits of the control value are used
  - if the input vectors contain `int`s or `float`s, all the control bits are used
  - the first two pairs of bits select elements from the first vector and the second two pairs of bits select elements from the second vector



AVX Shuffling

`_mm256_shuffle_ps/pd`	Select floating-point elements according to an 8-bit value
`_mm256_shuffle_epi8/` `_mm256_shuffle_epi32`	Select integer elements according to an 8-bit value
`_mm256_shufflelo_epi16/` `_mm256_shufflehi_epi16`	Select 128-bit chunks from two input vectors based on an 8-bit control value

- Note
  - `_mm256_shufflelo_epi16` and `_mm256_shufflehi_epi16` shuffle 16-bit values
    - as with `_mm256_shuffle_ps`, the control value is split into four pairs of bits that select from eight elements
    - but for `_mm256_shufflelo_epi16`, the 8 elements are taken from the 8 low 16-bit values
    - for `_mm256_shufflehi_epi16`, the 8 elements are taken from the 8 high 16-bit values

Example: Complex Multiplication

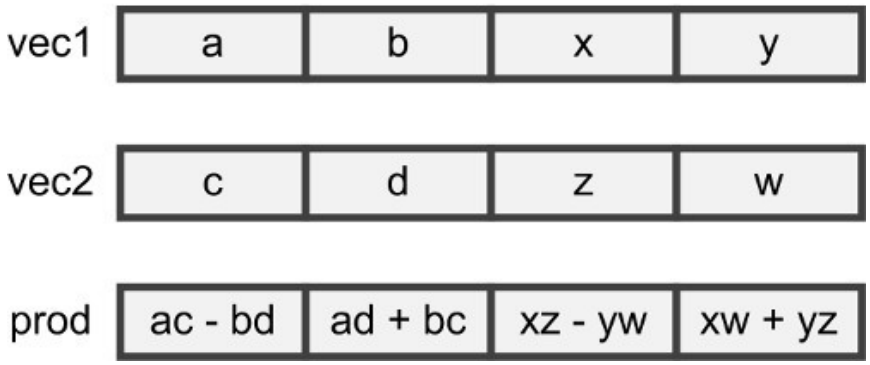
- Complex number: `a+bi`
  - real part: `a` and `b` are floating-point values
  - imaginary part: `i` is the square-root of -1
- Multiplication: `(a+bi)(c+di)` equals `(ac-bd)+(ad+bc)i`

Example: Complex Multiplication

- Assumptions: two complex multiplication at once
  - `vec1` is a `__m256d` that stores the two complex numbers `(a+bi)` and `(x+yi)`
  - `vec2` is a `__m256d` that stores `(c+di)` and `(z+wi)`
  - the `prod` vector stores the two products:
    - `(ac - bd) + (ad + bc)i`
    - `(xz - yw) + (xw + yz)i`

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

$$(x + zi)(y + wi) = xz - yw + (xw + yz)i$$



Example Solution

- Steps

1. multiply `vec1` and `vec2` and store the result in `vec3`
2. switch the real/imaginary values of `vec2`
3. negate the imaginary values of `vec2`
4. multiply `vec1` and `vec2` and store the result in `vec4`
5. use horizontal subtraction on `vec3` and `vec4` to produce the answer in `vec1`

```
#include <immintrin.h>
#include <stdio.h>
int main() {
  __m256d vec1 = _mm256_setr_pd(4.0, 5.0, 13.0, 6.0);
  __m256d vec2 = _mm256_setr_pd(9.0, 3.0, 6.0, 7.0);
  __m256d neg  = _mm256_setr_pd(1.0, -1.0, 1.0, -1.0);

  __m256d vec3 = _mm256_mul_pd(vec1, vec2);

  vec2 = _mm256_permute_pd(vec2, 0x5);

  vec2 = _mm256_mul_pd(vec2, neg);

  __m256d vec4 = _mm256_mul_pd(vec1, vec2);

  vec1 = _mm256_hsub_pd(vec3, vec4);

  double* res = (double*)&vec1;
  printf("%lf %lf %lf %lf\n", res[0], res[1], res[2], res[3]);

  return 0;
}
```

Lab Exercises

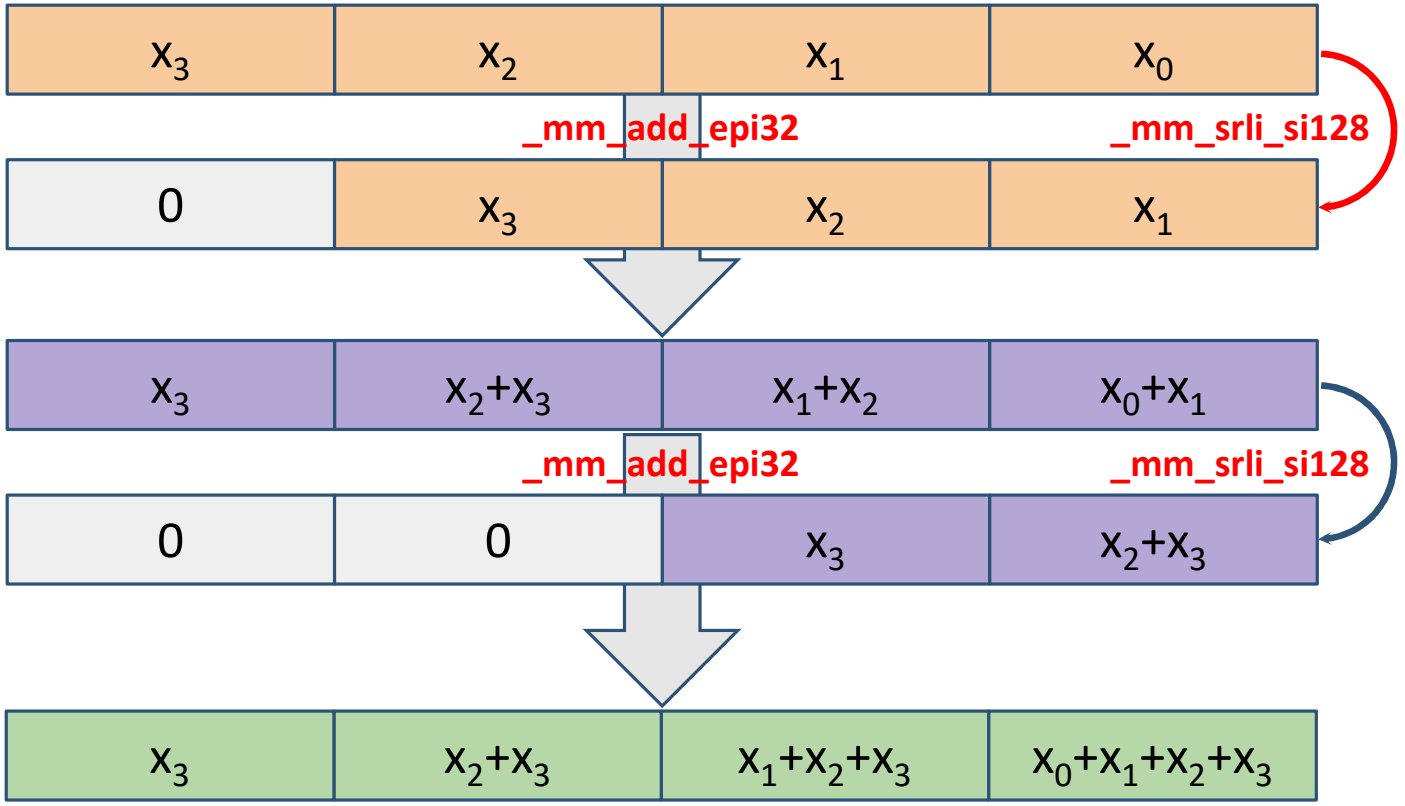
1. Multiplication of two array of complex numbers `(a+bi)(c+di)` in SoA layout
   - Remind: `(a+bi)(c+di)` equals `(ac-bd)+(ad+bc)i`

```
float a[SIZE];
float b[SIZE];
float c[SIZE];
float d[SIZE];
```

2. Prefix sum (cumulative sum)
   - Hints: `add` and `srli` (packed right shift)

```
uint32_t accumulator = 0;
for (size_t i = 0; i < n; i++) {
  accumulator += array[i];
  array[i] = accumulator;
}
```

SIMD Prefix Sum



Autovectorization and VLA Programming

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

- Automatic vectorization
  - Loop and SLP Vectorization
  - Autovectorization in LLVM
  - When the compiler fails
- Vector length agnostic ISA (ARM SVE)
- Vectorization in OpenMP

Automatic Vectorization

- Compilers can try to make use of SIMD instructions automatically
  - this optimization is also called autovectorization
  - automatic vectorization is a special case of automatic parallelization
    - a program is converted from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation, which processes one operation on multiple pairs of operands at once
- C/C++ compilers
  - `gcc` and `clang/llvm` will try to do autovectorization when you specify -O3 flag or other flags:
    - gcc: `-ftree-vectorize`
    - llvm: `-loop-vectorize`
- Two main strategies
  - LLV: Loop-level vectorization
  - SLP: Superword-level vectorization

Loop Level Vectorization

- Loop vectorization is a compiler transformation that tries to use vector instruction by exploiting (multiple) loop iterations
  - programs spend most of their time within such loops
  - therefore, vectorization can significantly accelerate them, especially over large data sets
  - implemented with vector instructions such as AVX2 or SVE
- Obstacles to vectorization
  - non-contiguous memory access
  - data dependencies
  - vectorization not efficient because of slow down, e.g., pipeline synchronization or data-movement

Loop Vectorization in LLVM: Scalar Code Example

```
void vec_mul(float a[], float b[], float c[], int size){
 for (int i=0; i<=size; i++){
    c[i] = a[i] * b[i];
 }
}
```

```
mov     dword ptr [rbp - 32], 0
.LBB0_1:  # =>This Inner Loop Header: Depth=1
mov     eax, dword ptr [rbp - 32]
cmp     eax, dword ptr [rbp - 28]
jg      .LBB0_4
mov     rax, qword ptr [rbp - 8]
movsxd  rcx, dword ptr [rbp - 32]
movss   xmm0, dword ptr [rax + 4*rcx]
                # xmm0 = mem[0],zero,zero,zero
mov     rax, qword ptr [rbp - 16]
movsxd  rcx, dword ptr [rbp - 32]
mulss   xmm0, dword ptr [rax + 4*rcx]
mov     rax, qword ptr [rbp - 24]
movsxd  rcx, dword ptr [rbp - 32]
movss   dword ptr [rax + 4*rcx], xmm0
mov     eax, dword ptr [rbp - 32]
add     eax, 1
mov     dword ptr [rbp - 32], eax
jmp     .LBB0_1
.LBB0_4:
```

- Compilation
  - compiler: x86-64 clang 11.0.0
    - flags: `-O0`
  - Function call code omitted
- Note
  - `movss/mulss:` scalar move and multi.
  - `xmm0` is a SSE vector register, here used scalarly

CompilerExplorer link https://godbolt.org/z/joE8Gvfn8

Loop Vectorization in LLVM: Vectorized Code Example

```
void vec_mul(float a[], float b[], float c[], int size){
 for (int i=0; i<=size; i++){
    c[i] = a[i] * b[i];
 }
}
```

- Compilation
  - compiler x86-64 clang 11.0.0
    - flags: `-O3 -fno-unroll-loops`
  - function call code omitted
  - loop tail omitted (see later)
- Note
  - `movups` move unaligned packed single precision floating point
  - `mulps` packed single precision floating point multiplication
  - `xmm` are SSE vector register
  - loop increment by 4

```
...

.LBB0_6: # =>This Inner Loop Header: Depth=1
movups  xmm0, xmmword ptr [rdi + 4*rax]
movups  xmm1, xmmword ptr [rsi + 4*rax]
mulps   xmm1, xmm0
movups  xmmword ptr [rdx + 4*rax], xmm1
add     rax, 4
cmp     rcx, rax
jne     .LBB0_6
cmp     rcx, r8
je      .LBB0_9
...

.LBB0_9:
```

Why unaligned?

CompilerExplorer https://godbolt.org/z/G5eb8Mq6d

Loop Tails

- Vectorization factor: number of iteration vectorized in a loop
- Example: AVX register (256bit), loop on float (32 bit) -> vector factor 8
- Problem: Loop size may not be a multiple of the vectorization factor

```
for (int i=0; i<=1024; i++){
   c[i] = a[i] * b[i];
}
```

vect. factor 8

```
for (int i=0; i<=128; i+=8){
   c[i] = a[i] * b[i];
}
```

```
for (int i=0; i<=1026; i++){
   c[i] = a[i] * b[i];
}
```

vect. factor 8

```
for (i=0; i<=128; i+=8){
   c[i] = a[i] * b[i];
}
for (; i<=1026; i++){
   c[i] = a[i] * b[i];
}
```

loop tail

Loop Tails in the Example

```
void vec_mul(float a[], float b[], float c[], int size){
 for (int i=0; i<=size; i++){
    c[i] = a[i] * b[i];
 }
}
```

- Vect. loop + loop tail

```
…
.LBB0_6: # =>This Inner Loop Header: Depth=1
movups  xmm0, xmmword ptr [rdi + 4*rax]
movups  xmm1, xmmword ptr [rsi + 4*rax]
mulps   xmm1, xmm0
movups  xmmword ptr [rdx + 4*rax], xmm1
add     rax, 4
cmp     rcx, rax
jne     .LBB0_6
cmp     rcx, r8
je      .LBB0_9
.LBB0_8:  # =>This Inner Loop Header: Depth=1
movss   xmm0, dword ptr [rdi + 4*rcx]   # xmm0 = mem[0],zero,zero,zero
mulss   xmm0, dword ptr [rsi + 4*rcx]
movss   dword ptr [rdx + 4*rcx], xmm0
add     rcx, 1
cmp     r8, rcx
jne     .LBB0_8
.LBB0_9:
ret
```

Vectorization with Non-contiguous Memory Access

- Four consecutive integers or floating-point values may be loaded directly from memory in a single SSE instruction
  - but if the four integers are not adjacent, they must be loaded separately using multiple instructions, which is considerably less efficient
  - vectorization is often possible but may be inefficient
- Examples
  - Stride 2
  - Inner loop with stride
  - Indirect addressing

```
for (int i=0; i<SIZE; i+=2)
  b[i] += a[i] * x[i];
```

```
for (int j=0; j<SIZE; j++)  {
  for (int i=0; i<SIZE; i++)
    b[i] += a[i][j] * x[j];
}
```

```
for (int i=0; i<SIZE; i+=2)
  b[i] += a[i] * x[index[i]];
```

Data Dependencies

- Sometime the compiler has insufficient information to decide to vectorize a loop
- Example

```
void copy(char *cp_a, char *cp_b, int n) {
  for (int i = 0; i < n; I++) {
    cp_a[i] = cp_b[i];
  }
}
```

- Do cp_a and cp_b overlap?
  - compiler must conservatively assume that the memory regions accessed by the pointer variables `cp_a` and `cp_b` may (partially) overlap
  - the compiler can generate a runtime test and generate both vectorial code and scalar code (like the tail we saw before)

Ignoring Dependencies

- Run-time data-dependency testing provides a generally effective way to exploit implicit parallelism
  - at the cost of slight increase in code size and testing overhead.
- If the function copy is only used in specific ways, however, you can assist the vectorizing compiler in two ways:

1. add an ivdeps pragma
2. restrict keyword
3. use the –fno-alias / -fargument-noalias

```
#pragma ivdep
void copy(char *cp_a, char *cp_b, int n) {
  for (int i = 0; i < n; i++) {
    cp_a[i] = cp_b[i];
  }
}
```

```
void copy(char * __restrict cp_a, char * __restrict cp_b, int n) {
  for (int i = 0; i < n; i++) cp_a[i] = cp_b[i];
}
```

Compiler Directives

Compiler hints (Intel ICC Compiler)	Semantics
`#pragma ivdep`	ignore data dependences
`#pragma vector always`	override efficiency heuristics
`#pragma novector`	disable vectorization
`__restrict__`	assert exclusive access through pointer
`__attribute__ ((aligned(int-val)))`	request memory alignment
`memalign(int-val,size);`	malloc aligned memory
`__assume_aligned(exp, int-val)`	assert alignment property

OpenMP Vectorization Support

- SIMD loop directives (OpenMP 4.0+)
  - can be applied to a loop to indicate that the loop can be transformed into a SIMD loop
  - multiple iterations of the loop can be executed concurrently using SIMD instructions

```
#pragma omp parallel for simd
for (i=0, i<N, i++){
  a[i] = N;
}
```

```
#pragma omp simd [clause]
for(…)
```

clause

```
if([simd :] scalar-expression)
safelen(length)
simdlen(length)
linear(list[ : linear-step])
aligned(list[ : alignment])
nontemporal(list)
private(list)
lastprivate0list)
reduction(reduction-identifier : list)
collapse(n)
order(concurrent)
```

OpenMP SIMD Examples

- Remove potential data dependencies that may hinder SIMD execution

```
#pragma omp simd safelen(4)
float A[SIZE], B[SIZE];
for(int i=4; i<SIZE; i++){
  A[i] = A[i-4] + B[i];
}
```

- Assert memory alignment properties

```
float x[SIZE], y[SIZE], A;
#pragma omp simd aligned(x:32, y:32)
for(int i=0; i<SIZE; i++){
  x[i] = A * x[i] + y[i];
}
```

SLP Vectorization

- SLP: Superword-Level Parallelism
  - combine similar independent instructions into vector instructions
  - vectorizes memory accesses, arithmetic operations, comparison operations, PHI-nodes
  - SLP-vectorizer processes the code bottom-up, across basic blocks, in search of scalars to combine
- Example
  - similar operations on its inputs (a1, b1) and (a2, b2)
  - the basic-block vectorizer may combine these into vector operations

```
void foo(int a1, int a2, int b1, int b2, int *A) {
  A[0] = a1*(a1 + b1);
  A[1] = a2*(a2 + b2);
  A[2] = a1*(a1 + b1);
  A[3] = a2*(a2 + b2);
}
```

ARM SVE

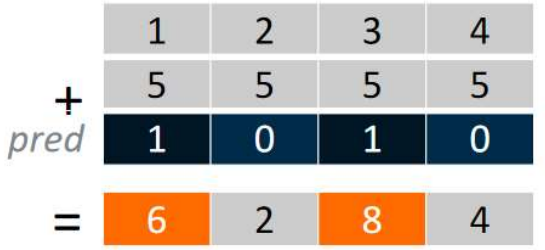
- Key architectural features
  - scalable vectors
  - per-lane predication
  - gather-load and scatter-store
  - speculative vectorization, horizontal and serialized vector operations
- Vector Length Agnostic (VLA) programming
  - traditional SIMD architectures defines a fixed size for their vector registers
  - SVE only specifies a maximum size
  - SVE programs can be vector-length agnostic: a single binary works on machines with different hardware vector lengths

Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis,Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, Paul Walker

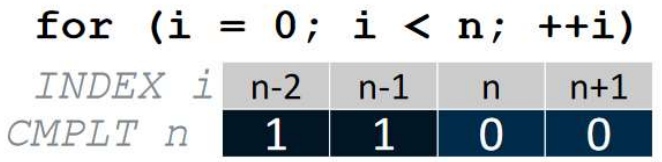
The ARM Scalable Vector Extension. IEEE Micro 37(2): 26-39 (2017)

SVE: Vector Length Agnostic Programming

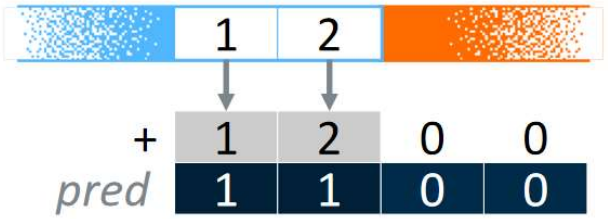
- Per-lane predication
  - operations work on individual lanes under control of a predicate register



- Predicate-driven loop control and management
  - eliminate scalar loop heads and tails by processing partial vectors

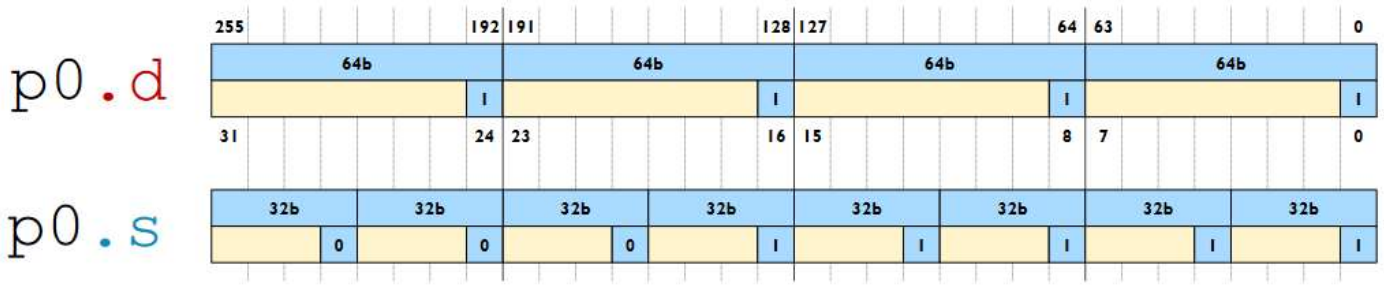


- Vector partitioning & software-managed speculation
  - First Faulting Load instructions allow memory accesses to cross into invalid pages



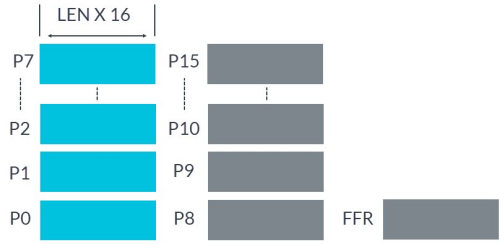
Predicates: Active Lanes vs Inactive Lanes

- Predicate registers track lane activity
  - 16 predicate registers (`p0-p15`)
  - 1 predicate bit per 8 vector bits (lowest predicate bit per lane is significant)
  - on load, active elements update the destination
  - on store, inactive lanes leave destination unchanged (`p0/m`) or set to 0's (`p0/z`)



SVE Registers

- SVE registers
  - 32 scalable vector registers, `z0-z31`
    - `z0-z31` can be implemented from 128 bits up to 2048 bits wide
      - the length: power of 2 and multiple of 128
    - the bottom 128 bits are shared with the fixed 128-bit `v0-v31` NEON vectors
    - can hold 64, 32, 16, and 8-bit elements
      - support integer, double-/single-/half-precision floating-point elements
  - 16 scalable predicate registers, `p0-p15`
    - used as bit masks for data operations
    - 1/8th size of SVE registers (1 bit / byte)
  - 1 First Fault predicate register (FFR)
    - indicate how successful the load and store operation for each element is
    - support speculative memory accesses
  - scalable vector system control registers `ZCR_Elx`



SVE: Loop Vectorization Example

- Compare: Fixed-length vs Vector-Length Agnostic
  - ACLE: Arm C Language Extensions (ARM's Intrinsic)
  - Assume array `a` of type `float`

Original Code

```
for(int i=0; i< N; ++i) {
  a[i] = 2.0 * a[i];
}
```

128-bit NEON Vectorization with ACLE

```
int i;

// vector loop
for(i=0; (i<N-3) && (N&~3); i+=4) {
  float32x4_t va = vld1q_f32(&a[i]);
  va = vmulq_n_f32(va, 2.0);
  vst1q_f32(&a[i], va)
}

// scalar loop tail
for(; i< N; ++i)
  a[i] = 2.0 * a[i];
```

SVE Vectorization with ACLE

```
svbool_t all = svptrue_b32();
svbool_t Pg;

for(int i=0;
 svptest_first(all,Pg=svwhilelt_b32(i,N));
 i += svcntw()
){
  svfloat32_t va= svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

SVE: Loop Vectorization Example

SVE Vectorization with ACLE

```
svbool_t all = svptrue_b32();
svbool_t Pg;
for(int i=0; svptest_first(all,Pg = svwhilelt_b32(i,N)); i += svcntw())
{
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

`svbool_t`	an array of 32-bit floats used as predicate
`svfloat32_t`	an array of 32-bit floats, the exact number is defined at runtime based on the SVE vector length
`svptrue_b32()`	return an all-true predicate for a b32 pattern
`svptest_first()`	return true if the first active element is true
`svwhilelt_b32()`	return a predicate in which element N is active if, for all values M in the range [0, N], adding M to the first input gives a value that is less than the second
`svcntw()`	count the number of 32-bit elements
`svld1()`	load values from memory and store the results in a vector. The vector elements have the same width as the loaded data
`svmul_x()`	multiply two integer inputs and return the low half of the result. Setting inactive to unknown.
`svst1()`	read elements from a vector and store them to memory. The vector elements have the same width as the stored data. No truncation.

SVE and Scalar Loop Tails: Fixed vs VLA Approach

```
void example_for(int *out, int *a, int *b, int N) {
  for (int i=0; i<N; i++) {
    out[i] = a[i] + b[i];
  }
}
```

- Example
  - 32-bit integer
  - 128- bit vector length
- VLA
  - no scalar loop tail

SVE Vectorization with ACLE

```
1 MOV w3, w3
2 MOV x9, #0
3
4 WHILELT p1.s, x9, x3
5 B.NONE .L_return
6
7 .L_loopStart:
8 LD1W z1.s, p1/Z, [x1, x9, LSL #2]
9 LD1W z2.s, p1/Z, [x2, x9, LSL #2]
10 ADD z1.s, p1/M, z1.s, z2.s
11 ST1W z1.s, p1, [x0, x9, LSL #2]
12 INCW x9
13 WHILELT p1.s, x9, x3
14 B.FIRST .L_loopStart
15
16 .L_return:
17 RET
```

128-bit NEON Vectorization with ACLE

```
1 AND w6, w3, 0xfffffffc
2 CBZ w6, .L_tail
3
4 .L_vector_loop:
5 LD1 {v0.4s}, [x1], #16
6 LD1 {v1.4s}, [x2], #16
7 ADD v5.4s, v0.4s, v1.4s
8 ST1 {v5.4s}, [x0], #16
9 SUB w6, w6, #4
10 CBNZ w6, .L_vector_loop
11
12 .L_tail:
13 AND w3, w3, #3
14 CBZ w3, .L_end
15
16 .L_scalar_loop:
17 LDRSW w9, [x1], #4
18 LDRSW w10, [x2], #4
19 ADD w11, w9, w10
20 STR w11, [x0], #4
21 SUB w3, w3, #1
22 CBNZ w3, .L_scalar_loop
23
24 .L_end:
25 RET
```

AVX Lab Exercices

- `matmul` from lecture 2

Version	Implementation	Running time	Relative speedup	Absolute speedup	GLOPS	% of peak
1	Python	21041.67	1.00	1	0.007	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.119	0.014
4	+ interchange	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	+ parallel loops	3.04	17.97	6 921	45.211	5.408
7	+ tiling	1.79	1.70	11 772	76.782	9.184

AVX Lab Exercices

Version	Implementation	Running time	Relative speedup	Absolute speedup	GLOPS	% of peak
1	Python	21041.67	1.00	1	0.007	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.119	0.014
4	+ interchange	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	+ parallel loops	3.04	17.97	6 921	45.211	5.408
7	+ tiling	1.79	1.70	11 772	76.782	9.184
8	Parallel dived and conquer	1.30	1.38	16 197	105 722	12.646
9	+ compiler vectorization	0.70	1.87	30 272	196 341	23 486
10	+ AVX instrinsics	0.39	1.76	53 292	352 408	41 677
11	Intel MKL	0.41	0.97	51 497	335 217	40 098

Version 10 is competitive with Intel's professionally engineered Math Kernel Library!

AVX Lab Exercises

1. Vectorize matmul using OpenMP simd pragmas
2. Vectorize matmul using intrinsics

Roofline Model

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

Outline

- Roofline model
  - arithmetic intensity
  - basic roofline model
  - extensions

The Roofline Model

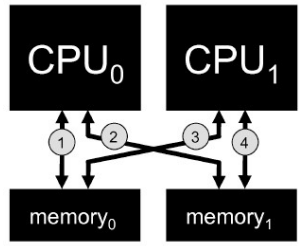
- Visually intuitive performance model constructed using bound and bottleneck analysis
- Abstract architectural model
  - it is designed to drive programmers towards an intuitive understanding of performance on modern computer architectures
  - not only provides programmers with realistic performance expectations, but also enumerates the potential impediments to performance
  - knowledge of these bottlenecks drives programmers to implement particular classes of optimizations

Samuel Webb Williams: Auto-tuning Performance on Multicore Computers
PhD Thesis. University of California, Berkeley. 2008

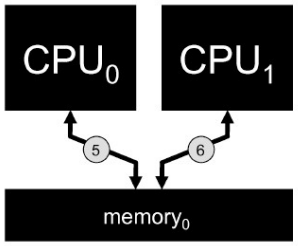
Samuel Williams, Andrew Waterman, David A. Patterson:
Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52(4): 65-76 (2009)

What determine performance?

- Depends on the architecture
  - Example: two different dual-processor architectures
    - conceptually, any processor can reference any memory location
    - however, bandwidth to a given processor may depend on which memory the address may lie in



Separate memories (NUMA)



Common shared memory

- Depends on the kernel (application)
  - Examples: different arithmetic intensities
    a) Dot products
    b) Stencil
    c) Matrix multiplication

```
temp=0.0;
for(i=0;i<N;i++){
  temp = A[i]*A[i];
}
magnitude = sqrt(temp);
```
0.125
(a)

```
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] );
}}
```
0.25
(b)

```
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];
}}}
```
N / 16
(c)

Understanding Arithmetic Intensity

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

0.125

(a)

```
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] );
}}
```

0.25

(b)

```
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];
}}}
```

N / 16

(c)

a) Dot product

Assumption: `N` is sufficiently large, does not fit in cache, large enough to amortize the poor performance of the square root. Performs `N` flops while transferring only $8 \cdot N$ bytes (`N` doubles), the second access to `A[i]` exploits the cache/register file locality within the processor. Arithmetic intensity = `0.125` flops per byte.

b) Stencil

Assumption: processor's cache is substantially larger than $8 \cdot N$, but substantially smaller than $16 \cdot N^2$. The leading point in the stencil `A[i,j+1]` will eventually be reused by subsequent stencils as `A[i+1,j]`, `A[i,j]`, `A[i-1,j]`, and `A[i,j-1]`. Although references to `A[i,j]` only generate $8 \cdot N^2$ bytes of communication, accesses to `C[i,j]` generate $16 \cdot N^2$ bytes because write-allocate cache architectures will generate both a read for the initial fill on the write miss in addition to the eventual write back. We get $6 \cdot N^2$ flops, therefore arithmetic intensity = $(6 \cdot N^2) / (24 \cdot N^2) = 0.25$

c) Dense matrix-matrix multiplication

Assumption: the cache is substantially larger than $24 \cdot N^2$, then `A[i,j]`, `B[i,j]`, and `C[i,j]` can be kept in cache and only their initial and write back references will generate DRAM memory traffic. Therefore, the loop nest will perform $2 \cdot N^3$ flops while only transferring $32 \cdot N^2$ bytes. The result is an arithmetic intensity = $N/16$

Performance

- There are three principal components to performance
  - Computation
  - Communication
  - Locality
- Each architecture has a different balance between these
- Each kernel has a different balance between these

- Performance is a question of how well a given kernel's characteristics map to an architecture's characteristics

Computation

- Common metric is floating point performance (Gflop/s), typically double precision
- Peak in-core performance can only be attained if
  - fully exploit ILP, DLP, FMA, etc…
  - non-FP instructions don't sap instruction bandwidth
  - threads don't diverge (GPUs)
  - transcendental/non pipelined instructions are used sparingly
  - branch mispredictions are rare
- To exploit a form of in-core parallelism, it must be
  - inherent in the algorithm
  - expressed in the high-level implementation
  - explicit in the generated code

Communication

- DRAM bandwidth (GB/s) is the metric of interest
- Peak bandwidth can only be attained if certain optimizations are employed
  - few unit stride streams
  - NUMA allocation and usage
  - software prefetching
  - on GPU, memory coalescing

- New wisdom: “Computation is free, communication is expensive”

Locality

- Maximize locality to minimize communication
  - there is a lower limit to communication: compulsory traffic
- Hardware changes can help minimize communication
  - larger cache capacities minimize capacity misses
  - higher cache associativities minimize conflict misses
  - non-allocating caches minimize compulsory traffic
- Software optimization can also help minimize communication
  - padding avoids conflict misses
  - blocking avoids capacity misses
  - non-allocating stores minimize compulsory traffic

Roofline Model

- Goal: integrate in-core performance, memory bandwidth, and locality into a single readily understandable performance figure
  - must graphically show the penalty associated with not including certain software optimizations
  - roofline model is unique to each architecture
  - coordinates of a kernel are ~unique to each architecture
  - built using a "bound and bottleneck" analysis
- Relates GB/s (bandwith) to GFlop/s (arithmetic intensity)
  - through dimensional analysis, its clear that Flops:Bytes is the parameter that allows us to convert bandwidth (GB/s) to performance (GFlop/s)
  - when we measure total bytes, we incorporate all cache behavior (the 3C's) and locality

The Roofline (1)

- Consider a simple kernel that must transfer B bytes of data from memory and perform F/2 floating-point operations on both CPU0 and CPU1
  - assume the memory can support PeakBandwidth bytes per second and combined, and the processors can perform PeakPerformance floating-point operations per second
- Simple analysis: suggests it will take B/PeakBandwidth seconds to transfer the data and F/PeakPerformance seconds to compute on it
  - assuming one may perfectly overlap communication and computation it will take

$$TotalTime = max \begin{cases} \dfrac{F}{PeakPerformance} \\ \dfrac{B}{PeakBandwith} \end{cases}$$

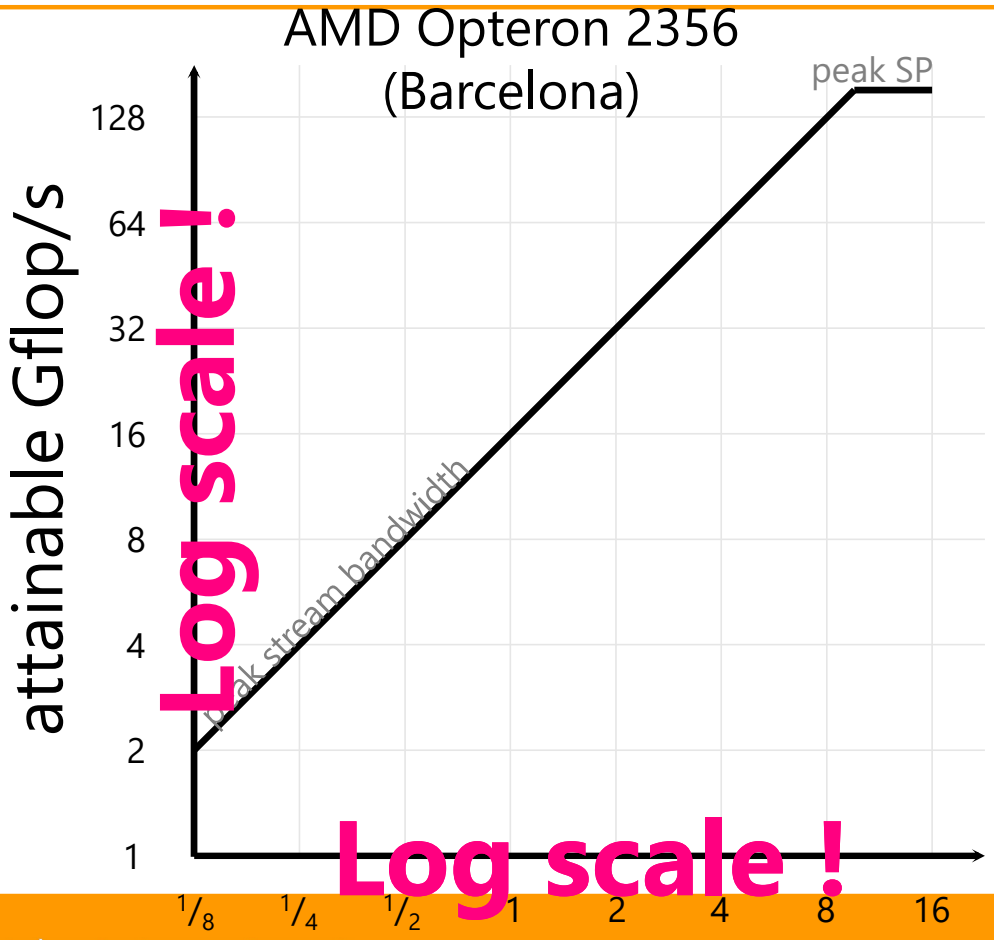
The Roofline (2)

- Reciprocating and multiplying by F flops, we observe performance is bound to:

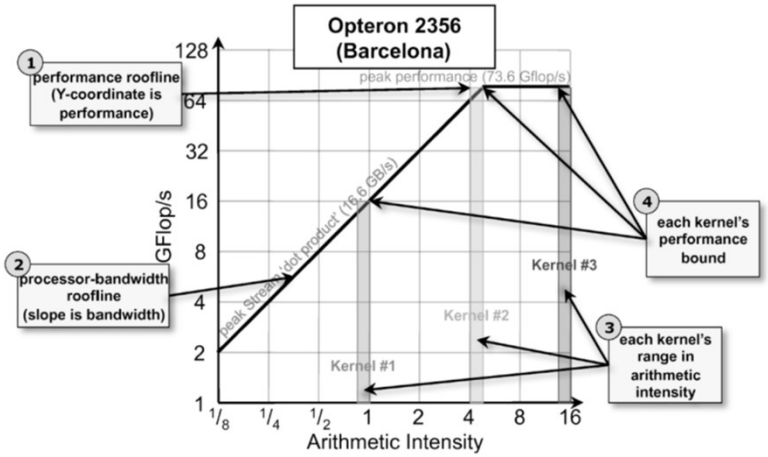
$$AttainablePerformance\ (Gflop/s) = min \begin{cases} PeakPerformance \\ PeakBandwidth\ \times\ AI \end{cases}$$

  - where AI (Arithmetic Intensity) is F/B, the actual flop:byte ratio
- Performance is upper bounded by both the peak flop rate, and the product of streaming bandwidth and the flop:byte ratio
- Note
  - bandwidth #'s collected via micro benchmarks
  - computation #'s derived from optimization manuals (pencil and paper)
  - assume complete overlap of either communication or computation

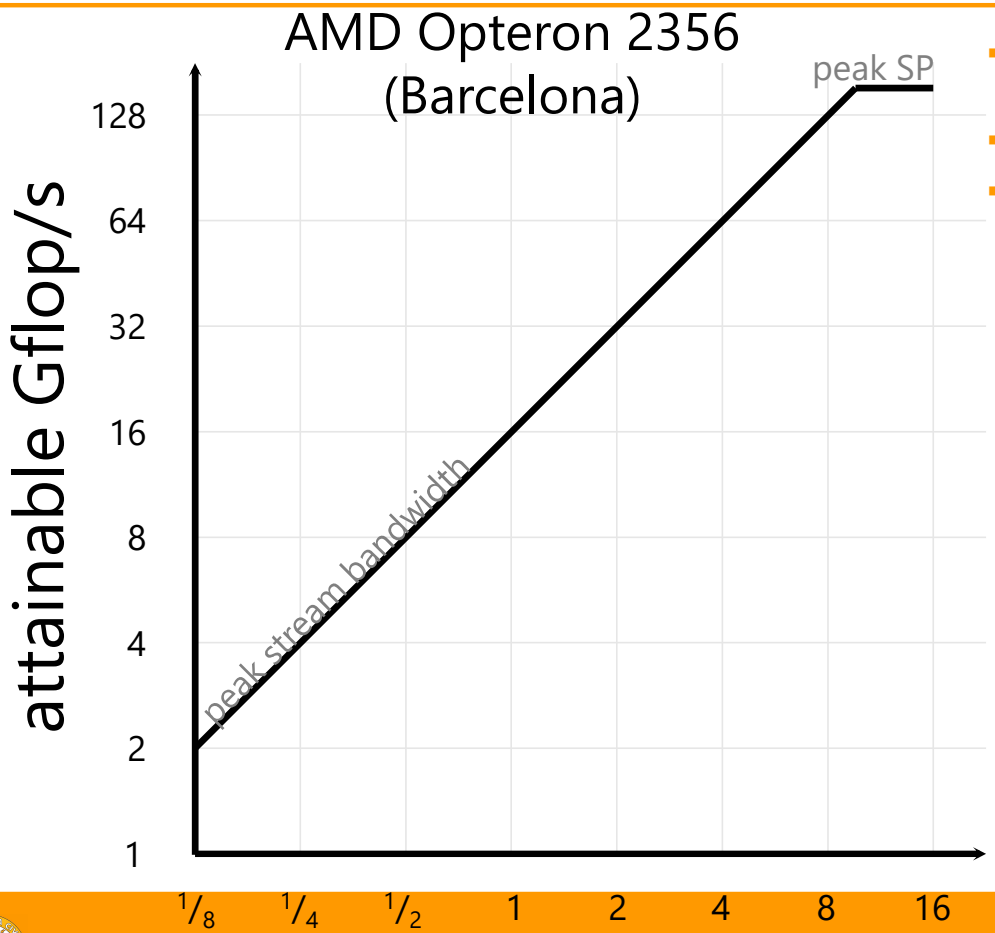
Roofline model for Opteron
(adding ceilings)



- Given the tremendous range in performance and arithmetic intensities, we will plot these figures on a log-log scale

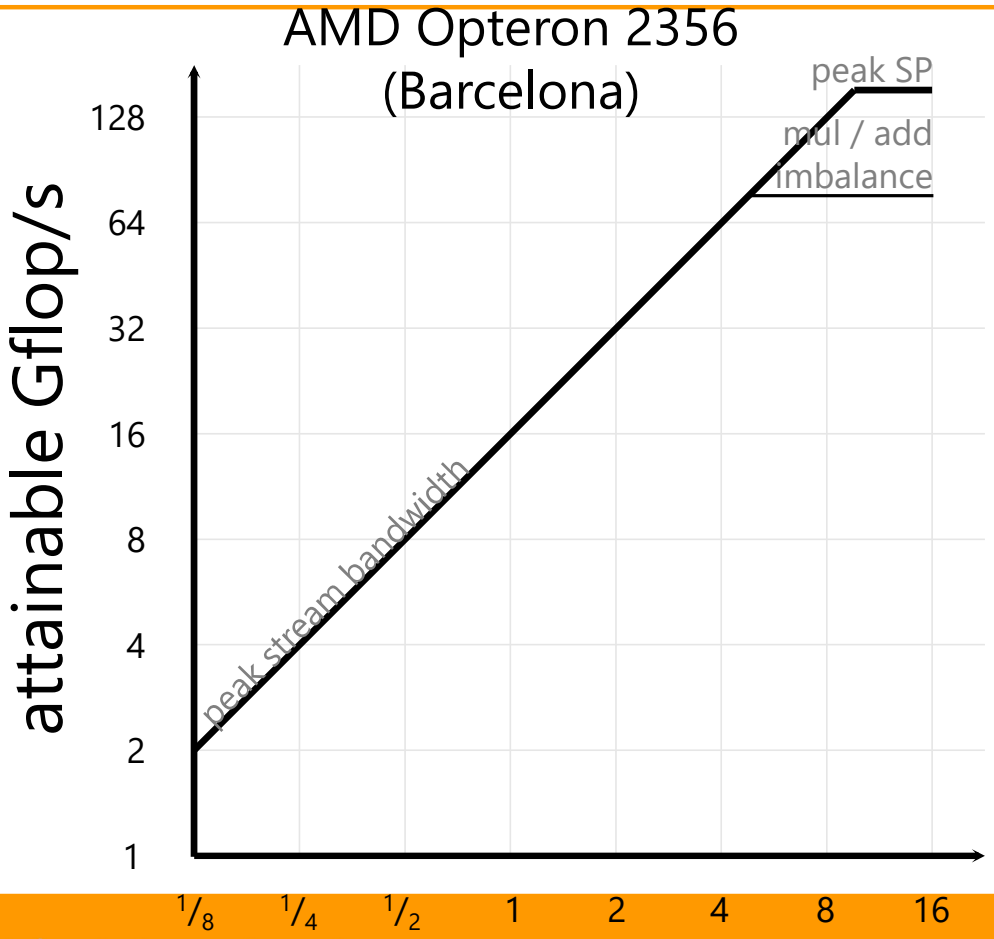


Roofline model for Opteron
(adding ceilings)



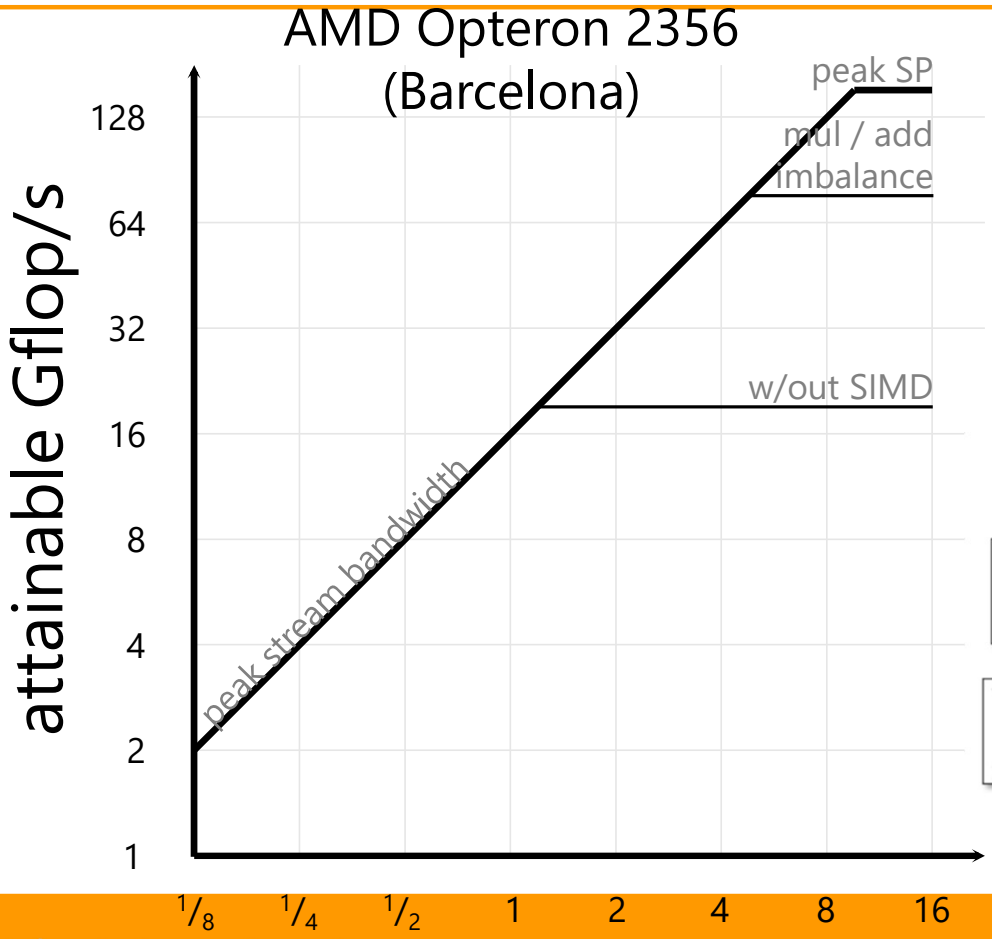
- Peak roofline performance based on single precision peak
- Bandwidth: hand tuned stream read for bandwidth
- As arithmetic intensity increases, so to does the performance bound
  - However, at the machine's flop:byte ratio, the performance bound saturates at the machine's peak performance
  - the slope of the roofline in the bandwidth-limited regions is actually the machine's stream bandwidth
  - on a log-log scale the line always appears at a 45-degree angle

Roofline model for Opteron
(adding ceilings)



- Opterons have separate multipliers and adders
  - 'functional unit parallelism'
- This is a ceiling beneath the roofline
- For codes that are dominated by one or the other, attainable performance will be half that of a code that has a perfect balance between multiplies and adds
- Examples:
  - PDEs on structured grids: often perform stencil operations which are dominated by adds with very few multiplies,
  - dense linear algebra often see a near perfect balance between multiplies and adds
- We may create a series of ceilings based on the ratio of adds to multiplies

Roofline model for Opteron
(adding ceilings)

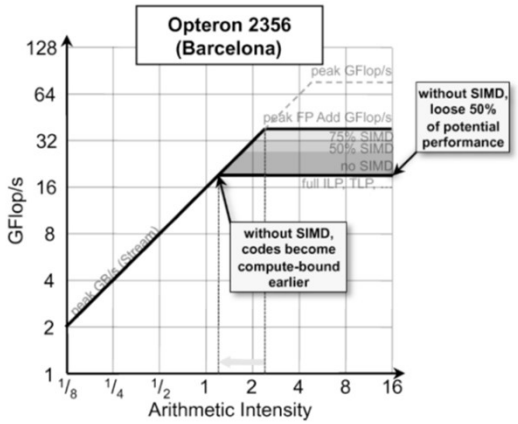


- In single precision, SIMD is 4x32b
- If only the `_ss` versions are used, performance is 1/4

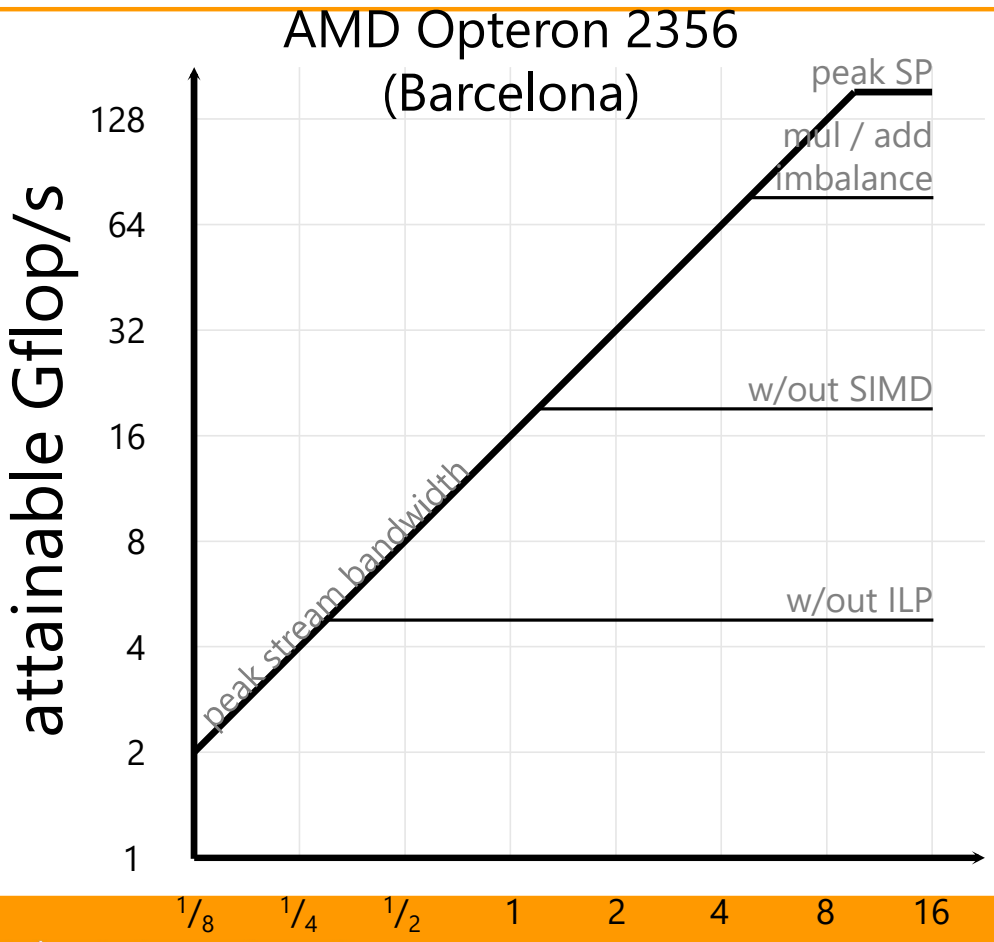
```
for(i=…){
 sum0+=b[i  ];
 sum1+=b[i+1];
 sum2+=b[i+2];
 sum3+=b[i+3];
}
```

```
for(i=…){
 sum0=_mm_add_sd(sum0,…b[i  ]…);
 sum1=_mm_add_sd(sum1,…b[i+1]…);
 sum2=_mm_add_sd(sum2,…b[i+2]…);
 sum3=_mm_add_sd(sum3,…b[i+3]…);
}
```

```
for(i=…){
 sum01=_mm_add_pd(sum01,…b[i  ]…);
 sum23=_mm_add_pd(sum23,…b[i+2]…);
 sum45=_mm_add_pd(sum45,…b[i+4]…);
 sum67=_mm_add_pd(sum67,…b[i+6]…);
}
```

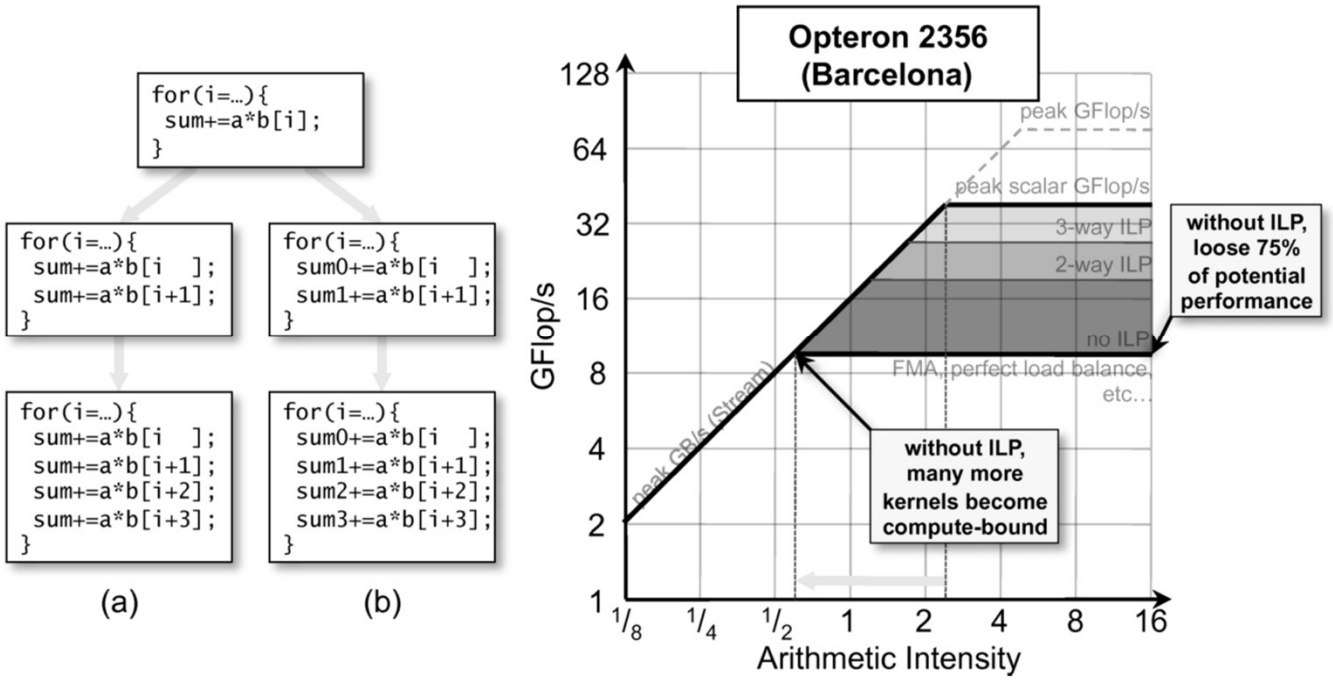


Roofline model for Opteron
(adding ceilings)



- If 4 independent instructions are kept in the pipeline, performance will fall
- How can we improve ILP?

Roofline model: ILP Optimization



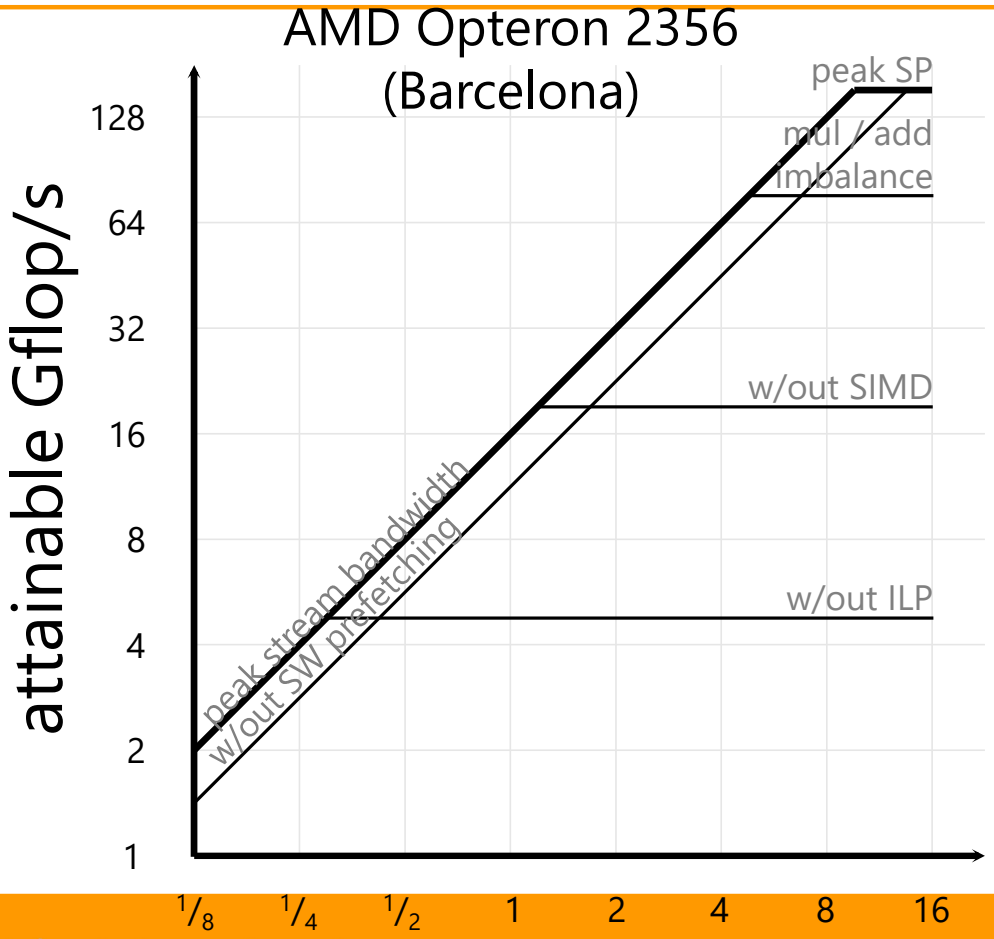
Performance ceilings as a result of insufficient instruction level parallelism.

(a) Code snippet in which the loop is unrolled. Note (FP add) instruction-level parallelism (ILP) remains constant.

(b) Code snippet in which partial sums are computed. ILP increases with unrolling.

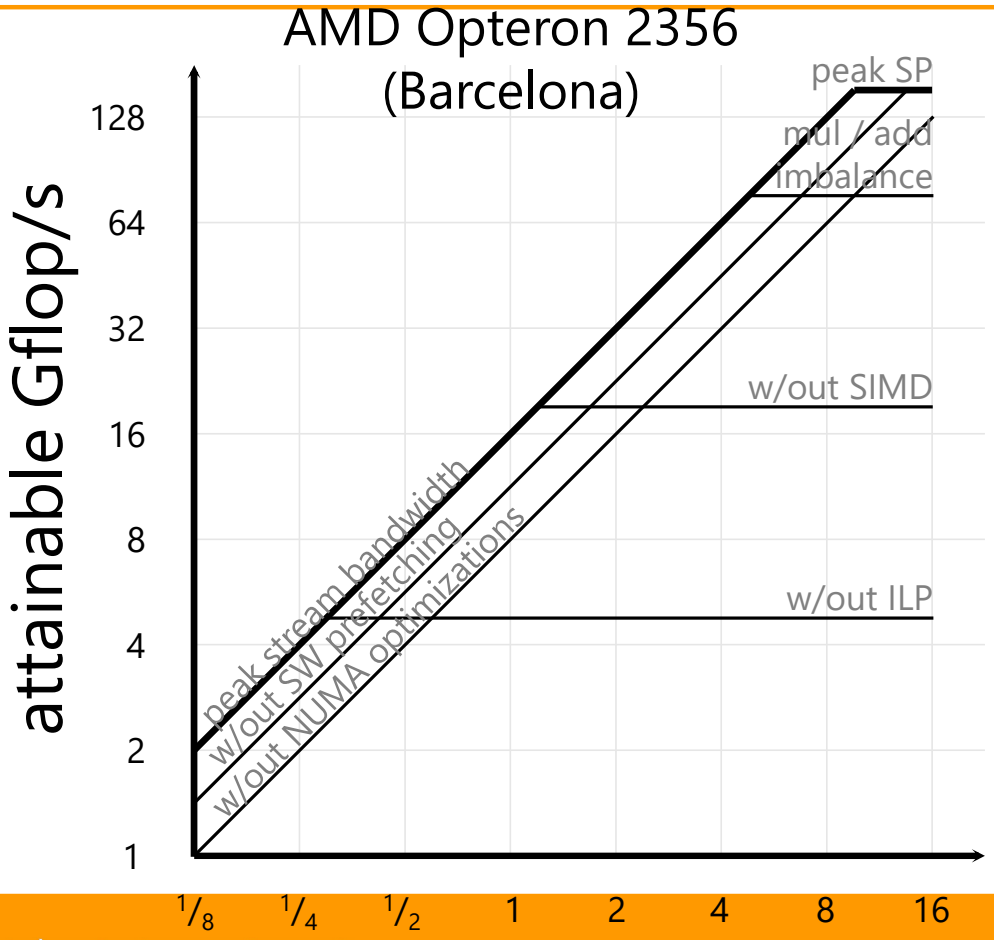
Roofline model for Opteron

(adding bandwidth ceilings)



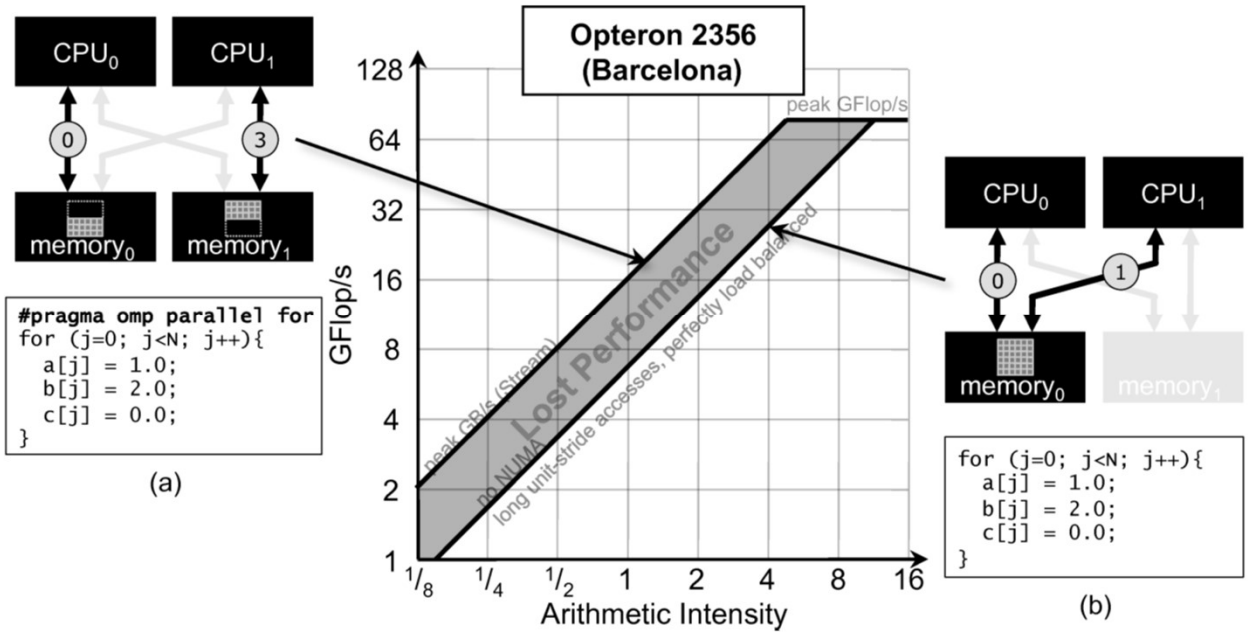
- If SW prefetching is not used, performance will degrade
- These act as ceilings below the bandwidth roofline

Roofline model for Opteron
(adding ceilings)



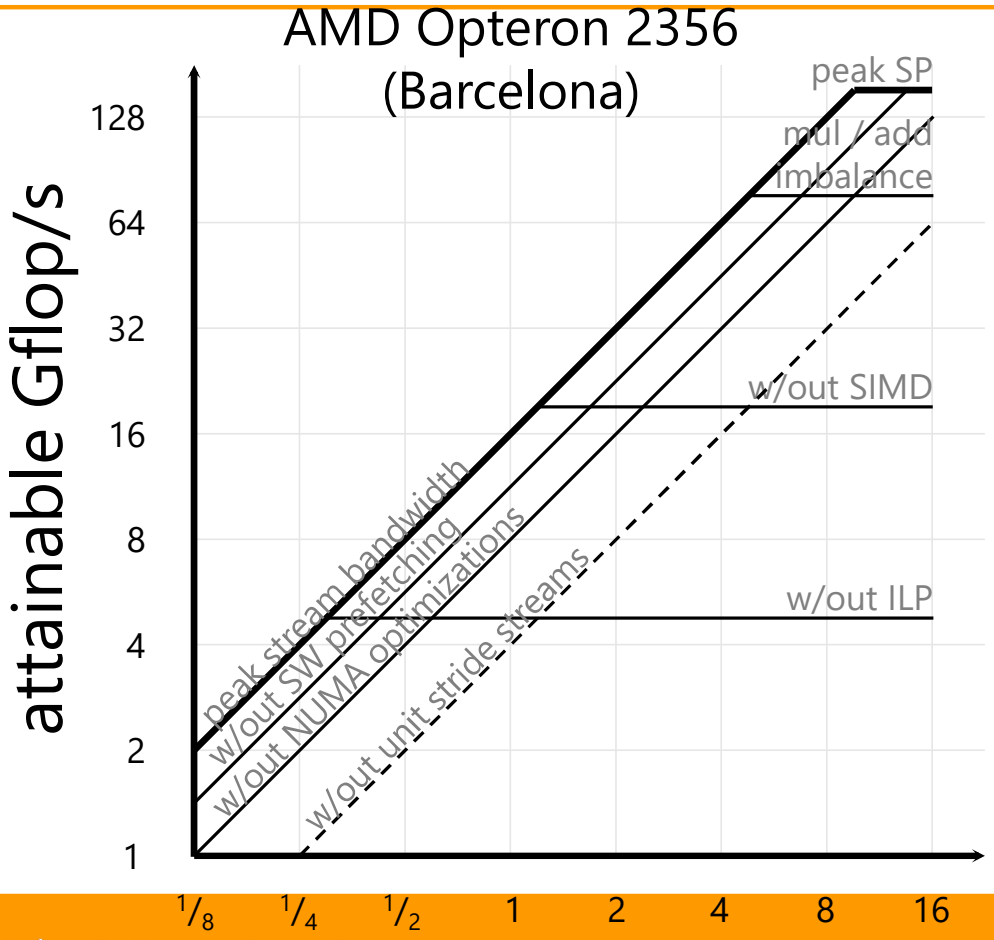
- Without NUMA optimizations, the memory controllers on the second socket can't be used

Roofline model: NUMA Ceiling



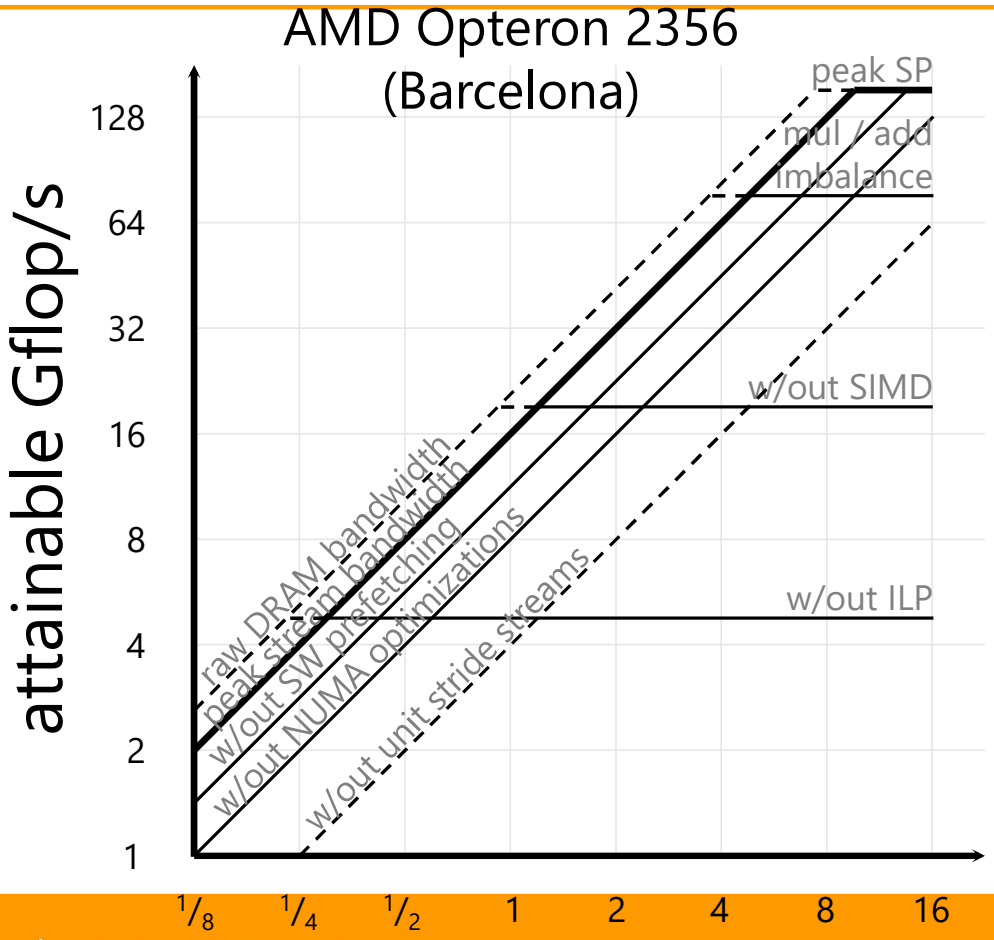
NUMA ceiling resulting from improper data layout. The codes shown are initialization-only (a) with and (b) without proper exploitation of a first-touch policy. Initialization is completely orthogonal to the possible computational kernels.

Roofline model for Opteron
(adding ceilings)



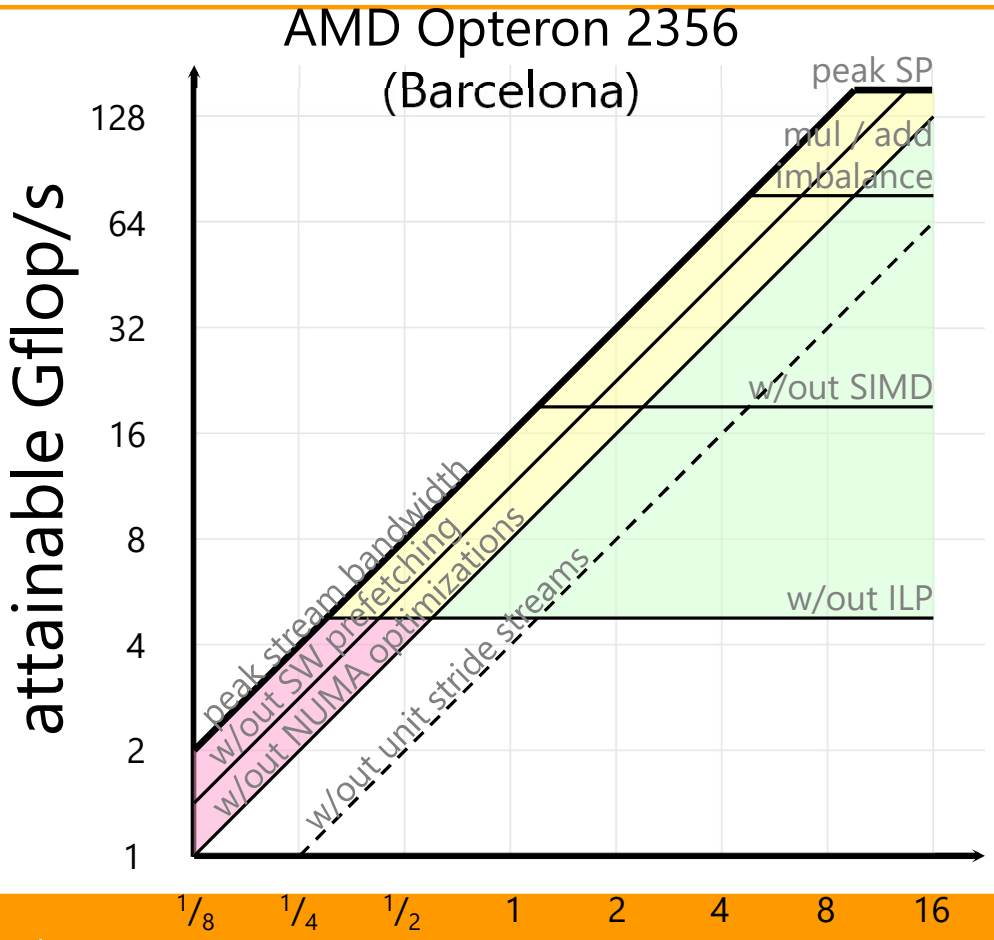
- Bandwidth is much lower without unit stride streams

Roofline model for Opteron
(adding ceilings)



- Its difficult for any architecture to reach the raw DRAM bandwidth

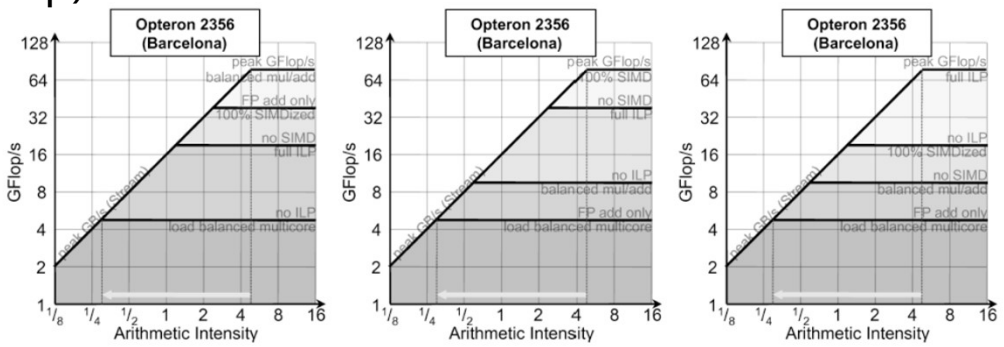
Roofline model for Opteron
(adding ceilings)



- Partitions the regions of expected performance into three optimization regions:
  - Compute only
  - Memory only
  - Compute+Memory

Uniqueness

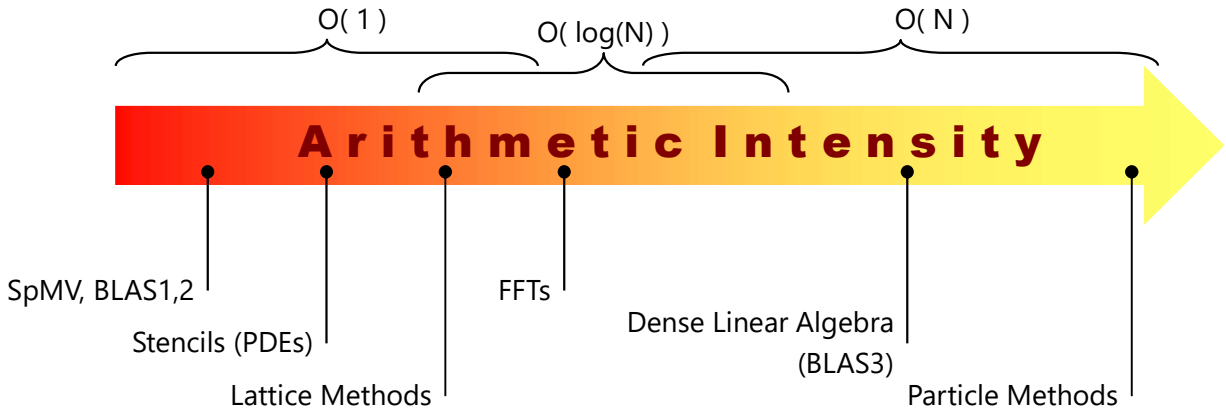
- There is no single ordering or roofline model
- The order of ceilings is generally (bottom up)
  - What is inherent in algorithm
  - What a compiler is likely to provide
  - What a programmer could provide
  - What can never be exploited for this kernel



- For example
  - FMA or mul/add balance is inherent in many linear algebra routines and should be placed at the bottom
  - However, many stencils are dominated by adds, and thus the multipliers and FMA go underutilized

Arithmetic Intensity in HPC

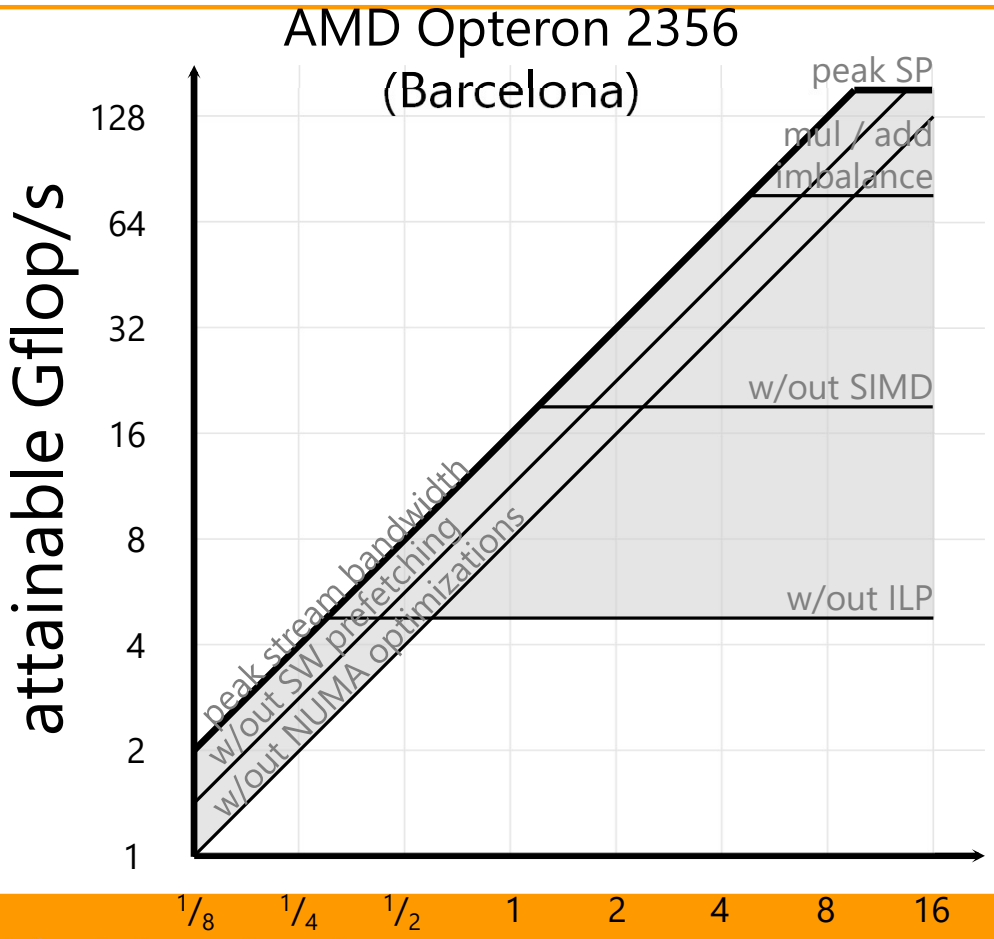
- Arithmetic Intensity (AI) ~ Total Flops / Total DRAM Bytes
  - some HPC kernels have an arithmetic intensity that's constant, but on others it scales with problem size (increasing temporal locality)
  - actual arithmetic intensity is capped by cache/local store capacity



Accurately Determining the true Flop:DRAM Byte ratio

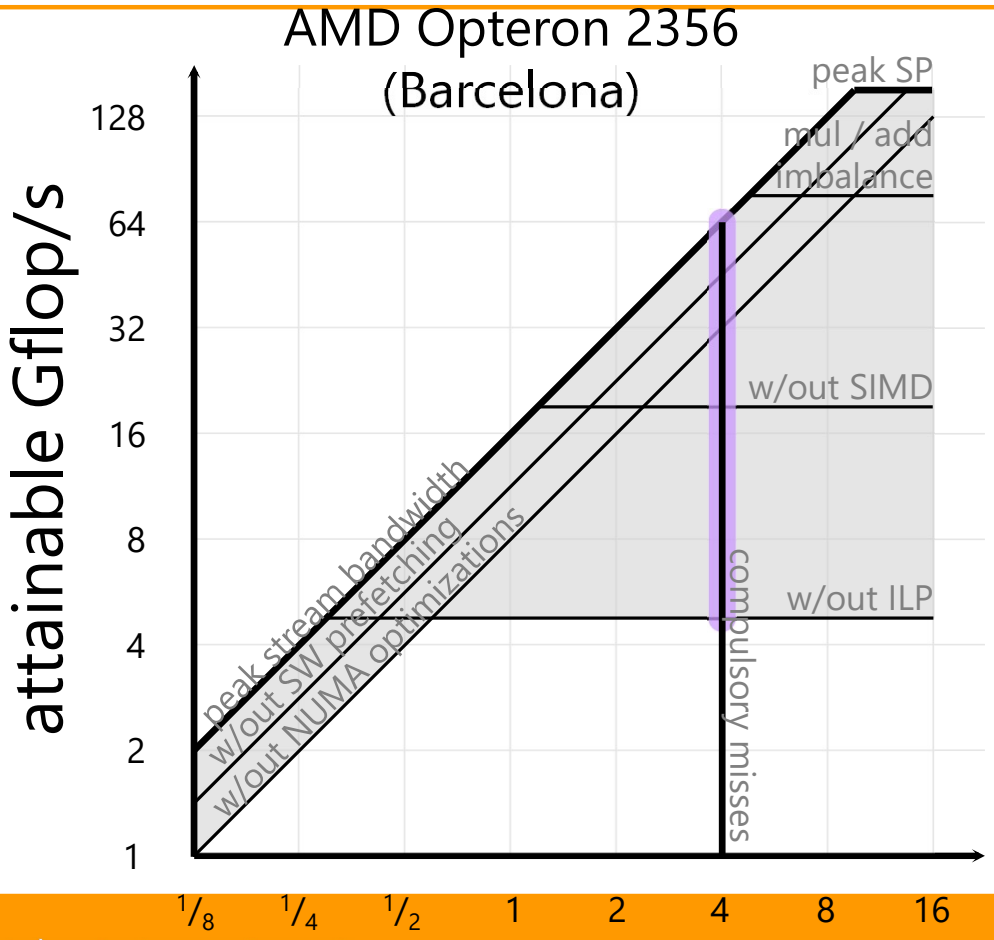
- Remember the 3C's of caches
- Calculating the Flop:DRAM byte ratio is:
  - Compulsory misses: straightforward
  - Capacity misses: pencil and paper (maybe performance counters)
  - Conflict misses: must use performance counters
- Flop:actual DRAM Byte ratio < Flop:compulsory DRAM Byte ratio
- One might place a range on the arithmetic intensity ratio
  - thus performance is limited to an area between the ceilings and between the upper (compulsory) and lower bounds on arithmetic intensity

Roofline model for Opteron
(powerpoint doodle)



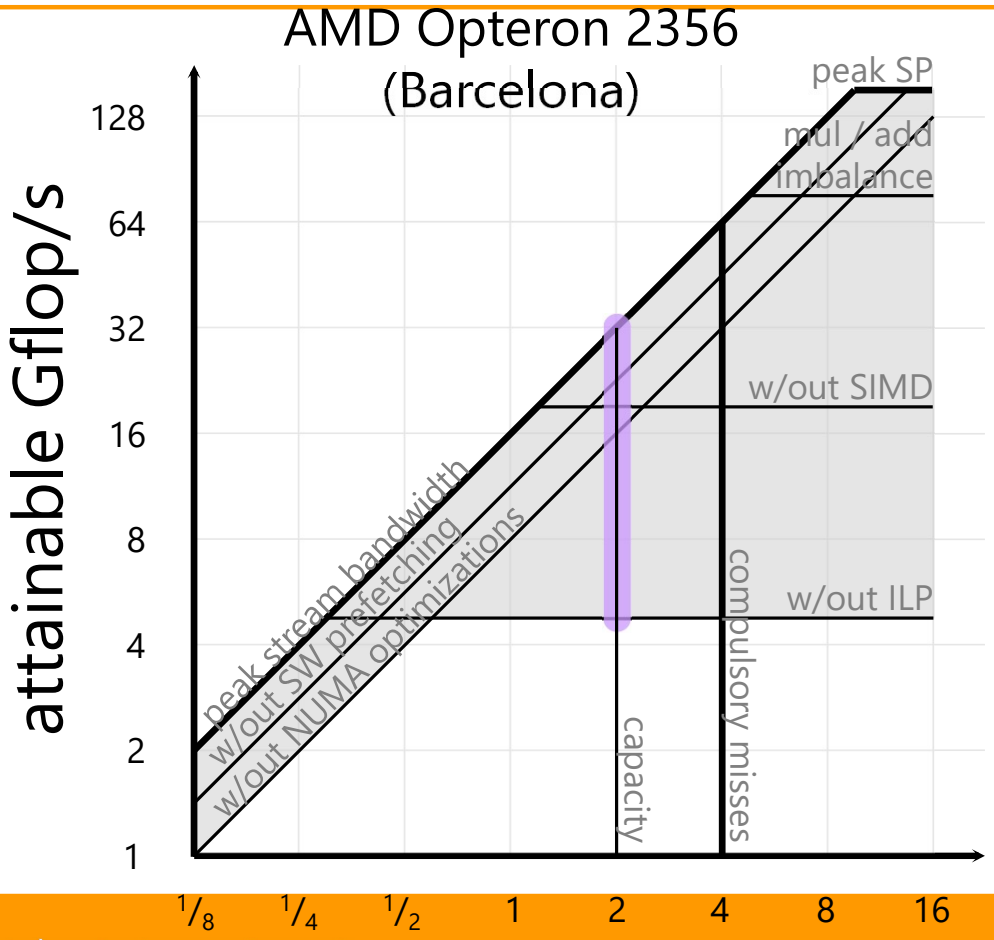
- Final Roofline

Roofline model for Opteron
(powerpoint doodle)



- Some arbitrary kernel has a flop:compulsory byte ratio of 4
- Overlaid on the roofline
- Defines upper bound on range of expected performance
- Also shows which optimizations are likely

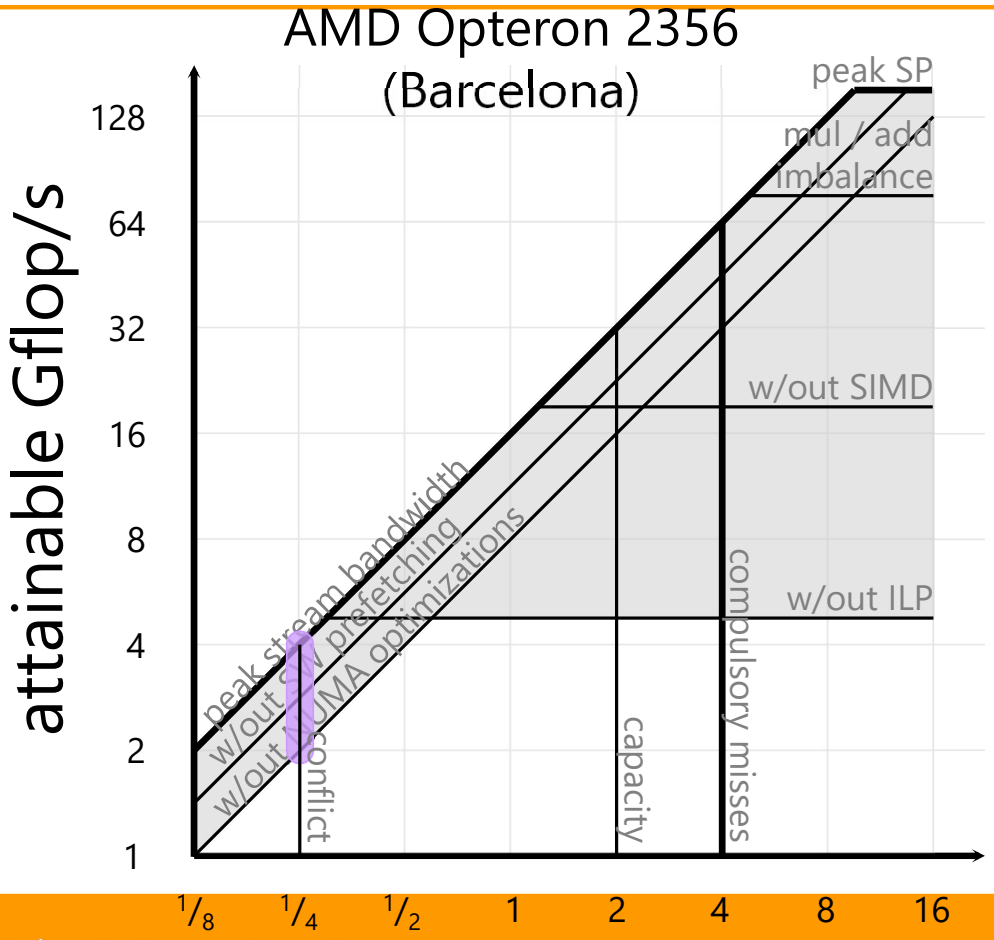
Roofline model for Opteron
(powerpoint doodle)



- Capacity misses reduce the actual flop:byte ratio
- Also reduces attainable performance
- AI is unique to each combination of kernel and architecture

- Capacity misses occur when the amount of data referenced by a program exceeds the capacity of the cache, requiring that some data be evicted to make room for new data. If the evicted data is referenced again by the program, a cache miss occurs, which is termed a capacity miss

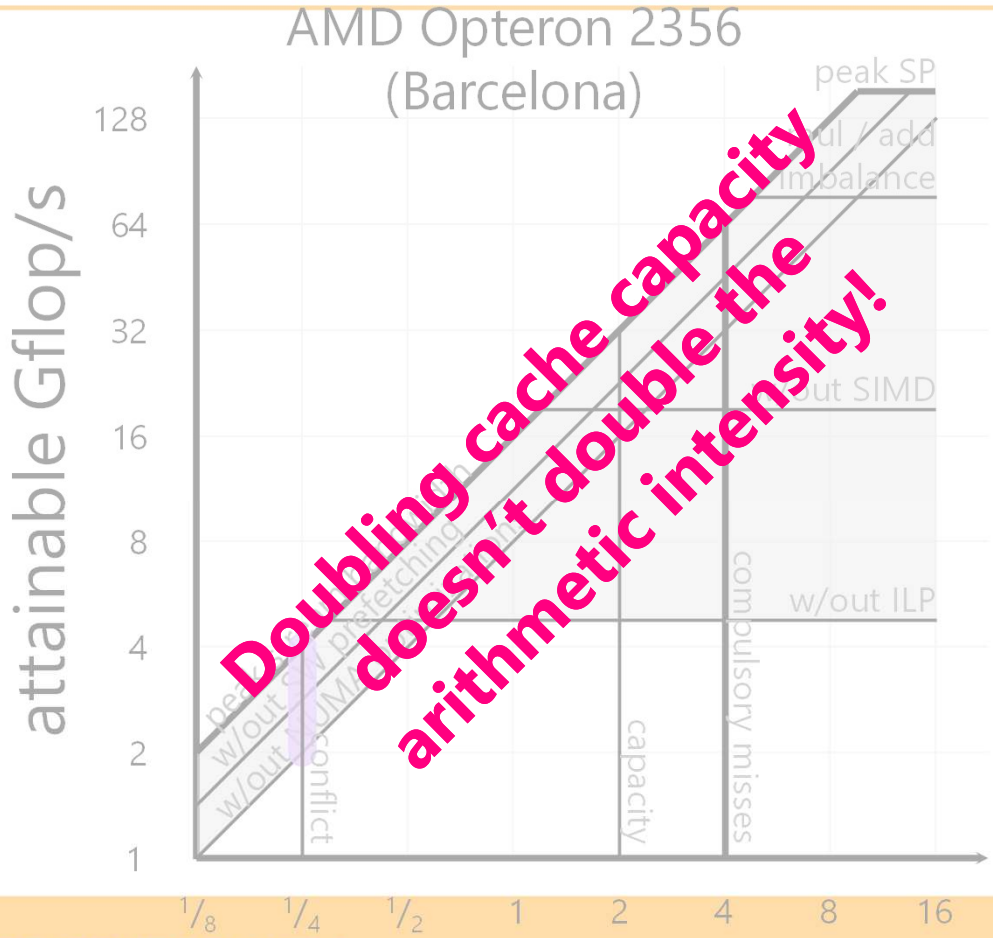
Roofline model for Opteron
(powerpoint doodle)



- Conflict misses may destroy performance
- AI is unique to each combination of kernel and architecture

- Conflict misses occur when a program references more lines of data that map to the same set in the cache than the associativity of the cache, forcing the cache to evict one of the lines to make room. If the evicted line is referenced again, the miss that results is a conflict miss

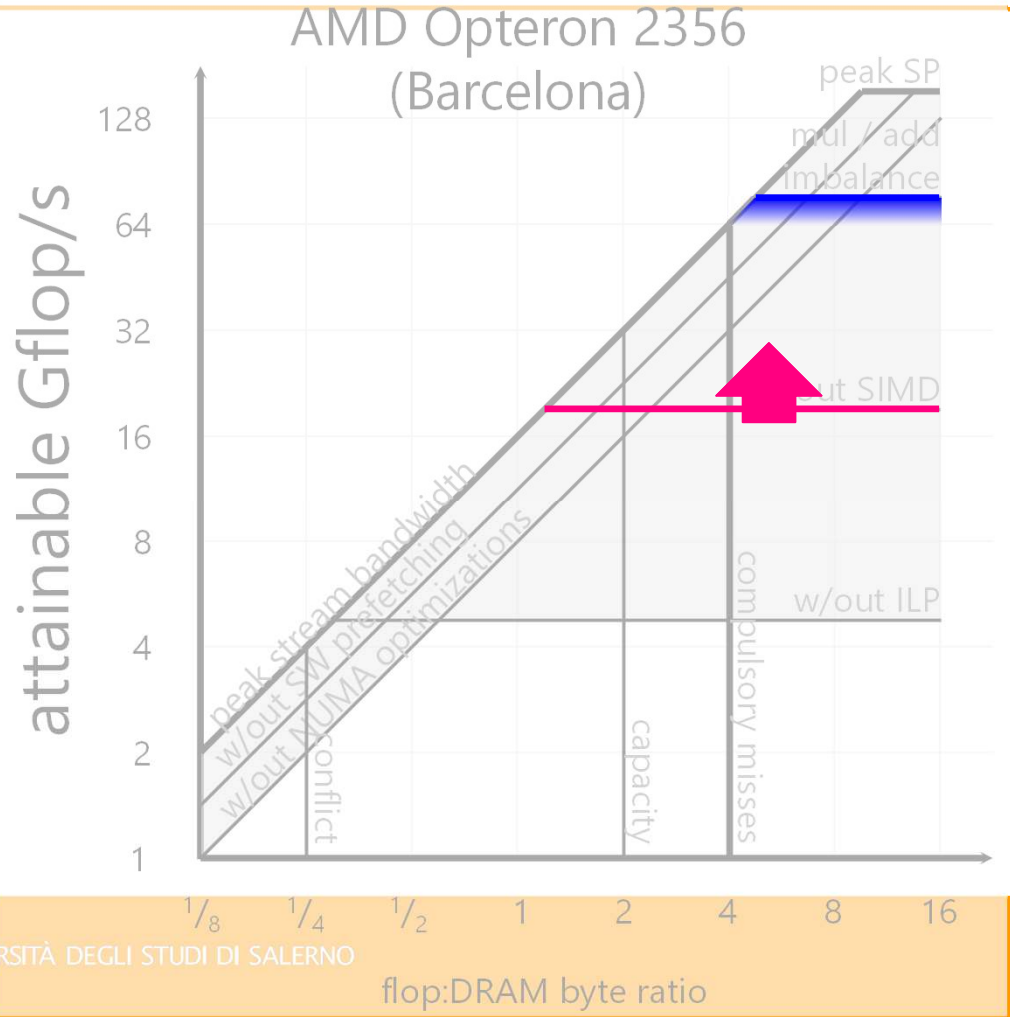
Roofline model for Opteron
(powerpoint doodle)



- Conflict misses may destroy performance

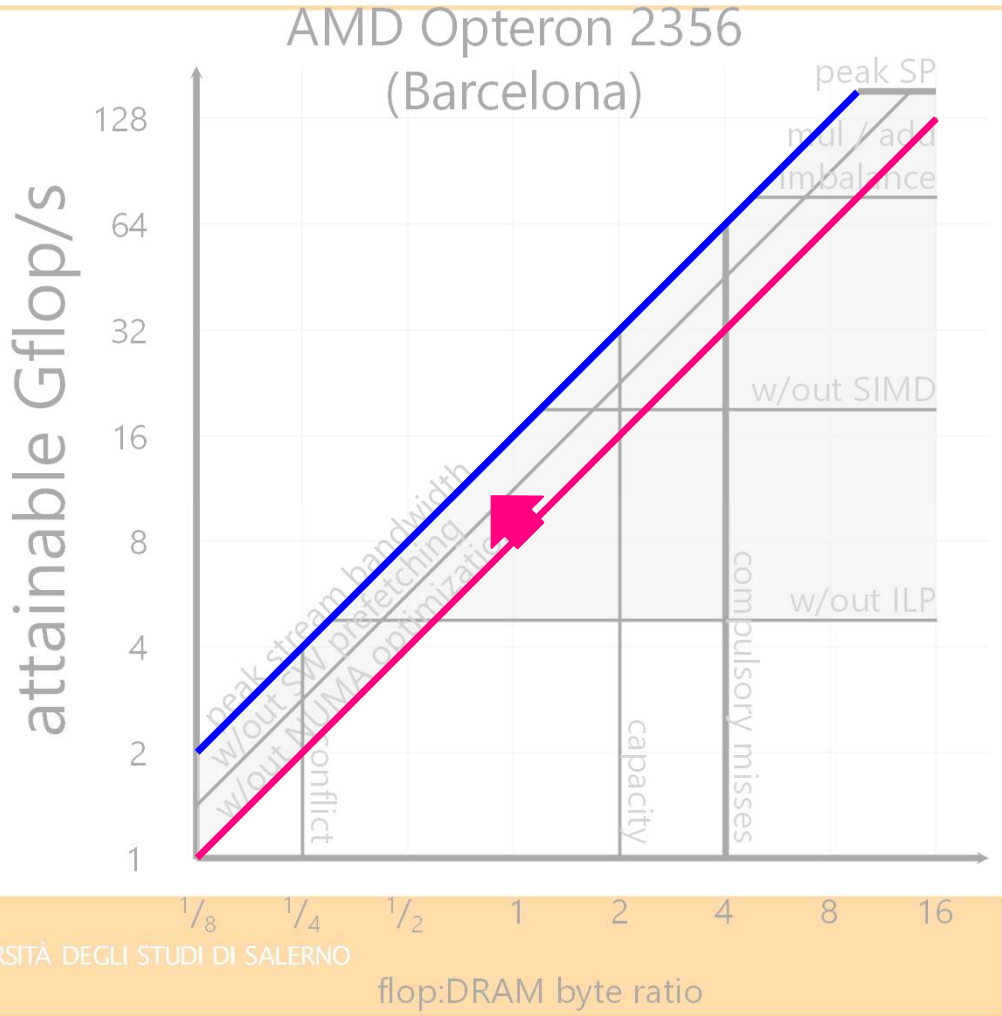
Doubling cache capacity doesn't double the arithmetic intensity!

Maximizing Attained in-core Performance



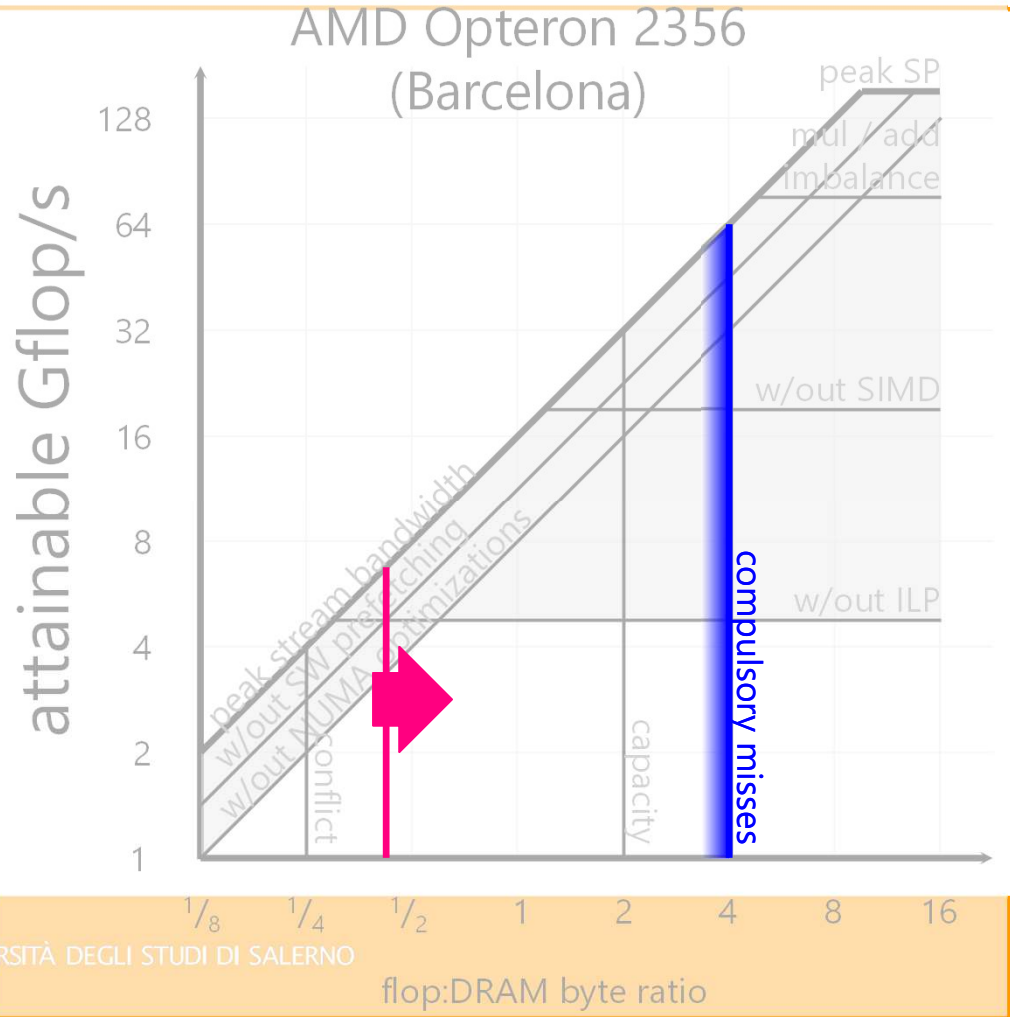
- Software optimizations such as explicit SIMDization can punch through the horizontal ceilings (what can be expected from a compiler)
- Other examples include loop unrolling, reordering, and long running loops

Maximizing Attained Memory Bandwidth



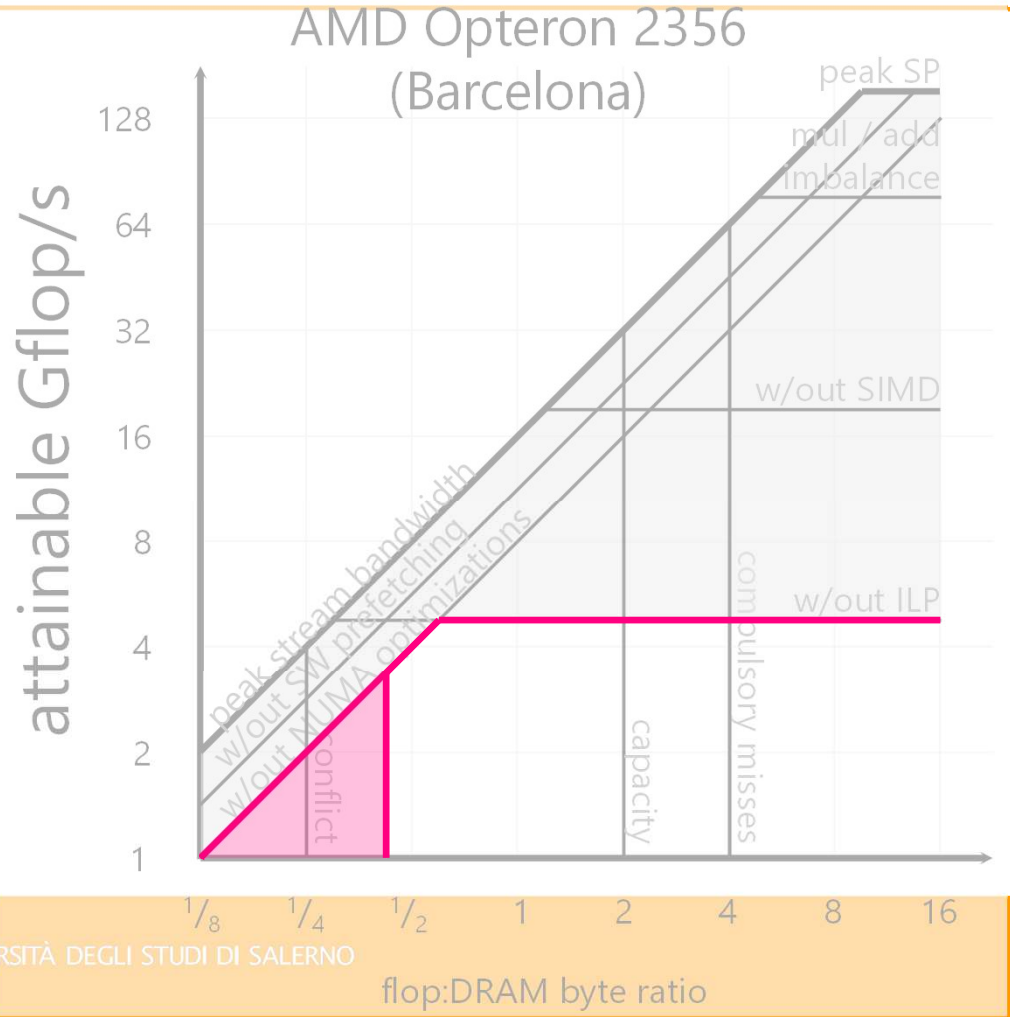
- Compilers won't give great out-of-the box bandwidth
- Punch through bandwidth ceilings:
  - Maximize MLP
  - long unit stride accesses
  - NUMA aware allocation and parallelization
  - SW prefetching

Minimizing Memory Traffic



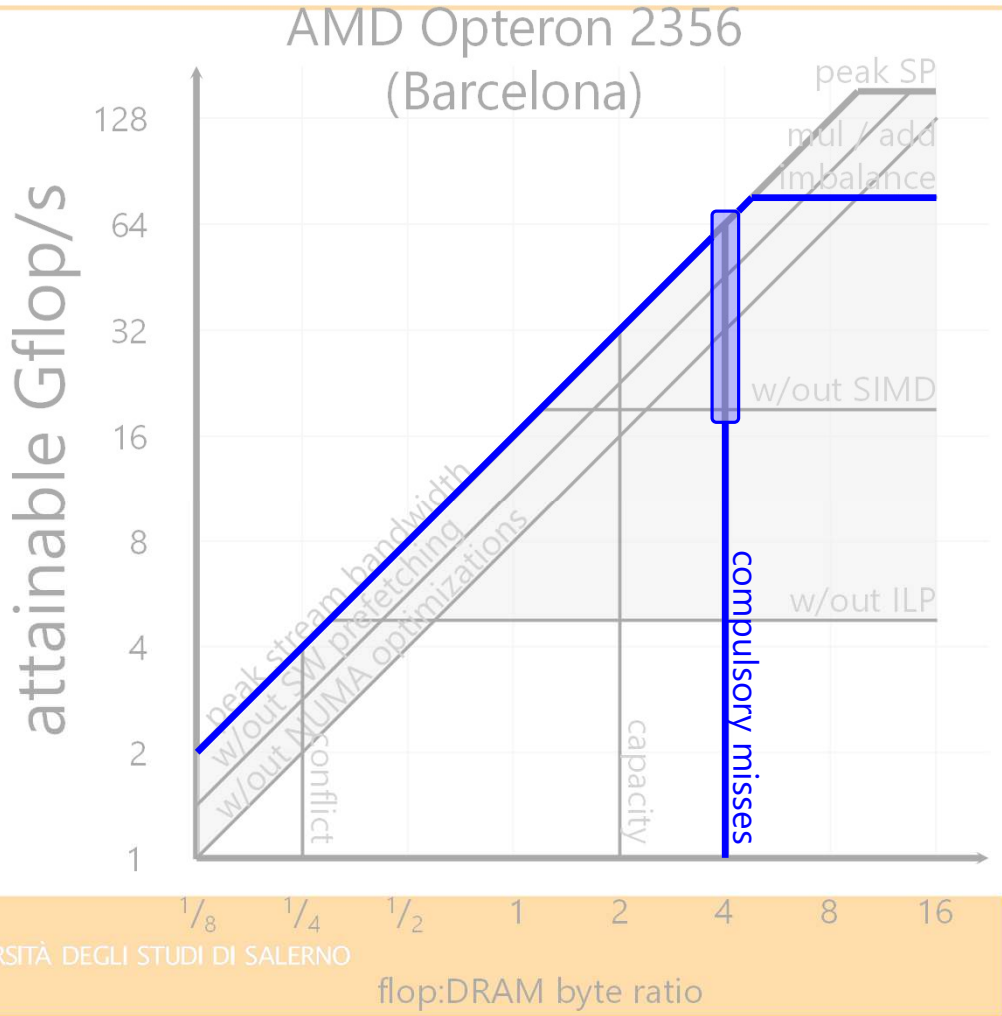
- Use performance counters to measure flop:byte ratio (AI)
- Out-of-the-box code may have an AI ratio much less than the compulsory ratio
  - Be cognizant of cache capacities, associativities, and threads sharing it
  - Pad structures to avoid conflict misses
  - Use cache blocking to avoid capacity misses
- These optimizations can be imperative

Effective Roofline (before)



- Before optimization, traffic, and limited bandwidth optimization limits performance to a very narrow window

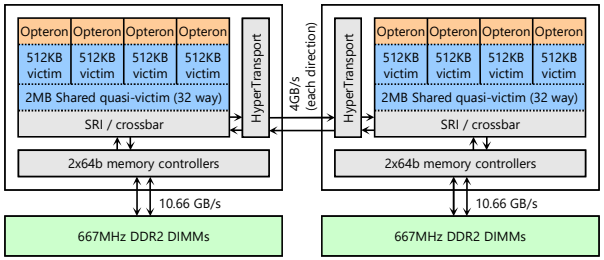
Effective Roofline (after)



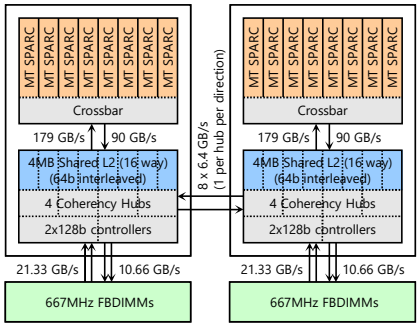
- After optimization, ideally, performance is significantly better

Four Architectures

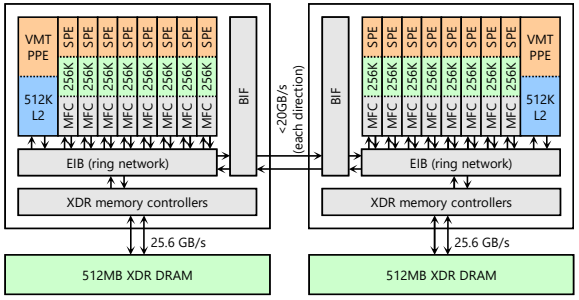
AMD Barcelona



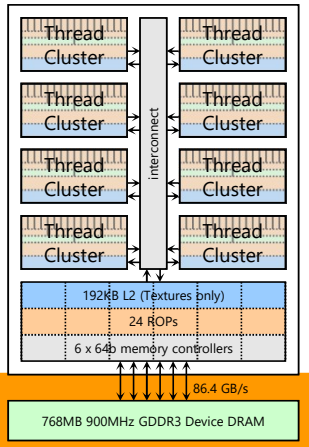
Sun Victoria Falls



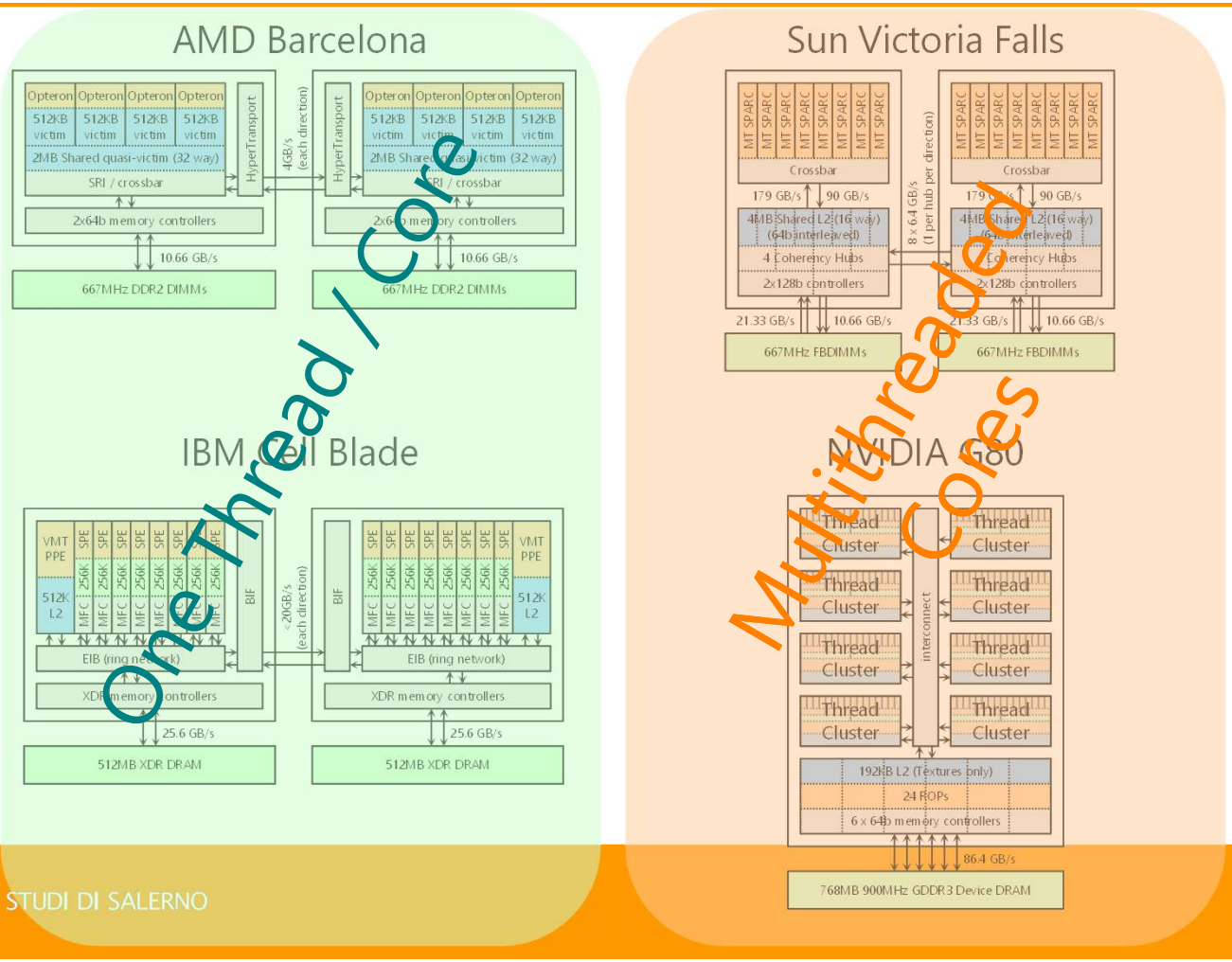
IBM Cell Blade



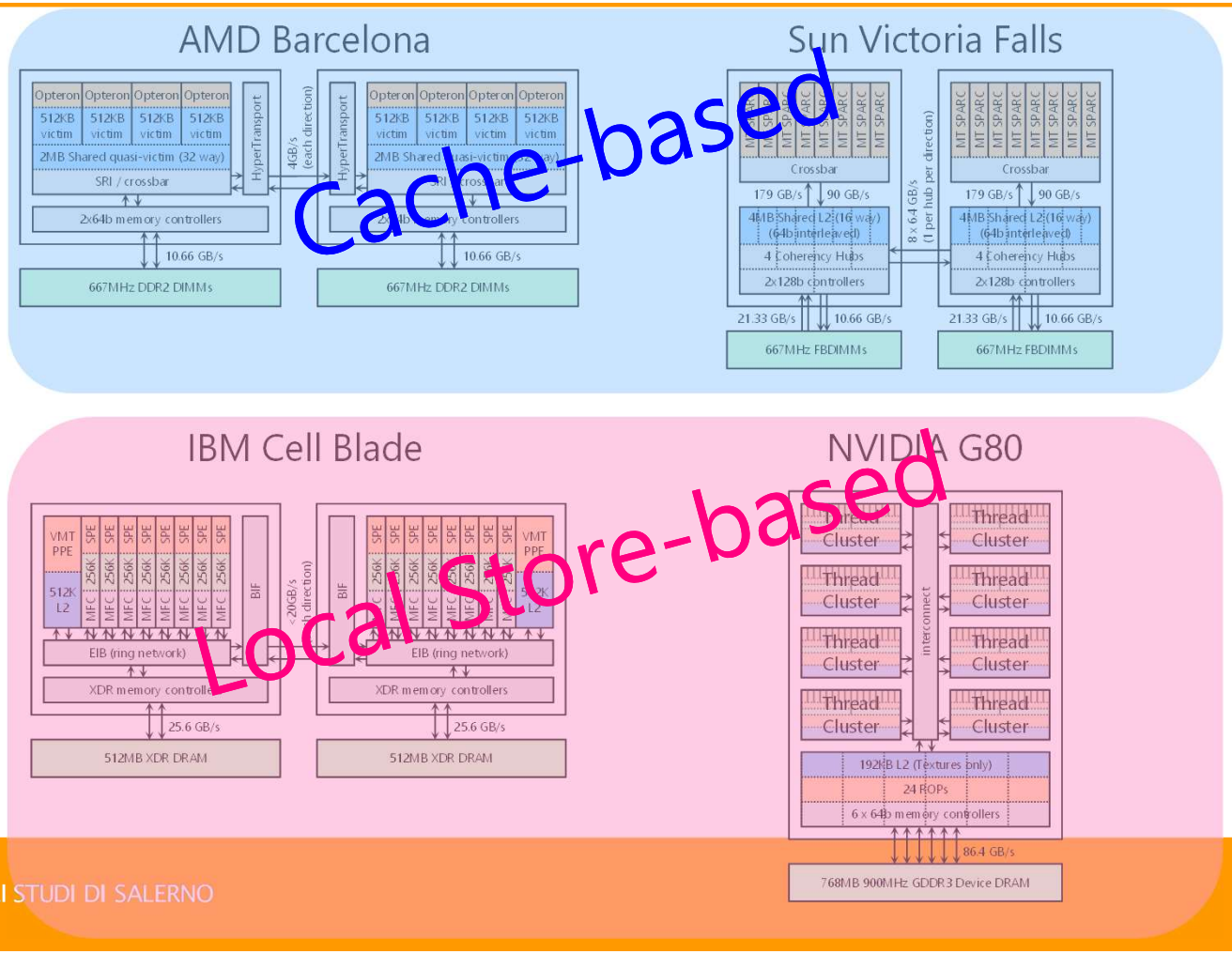
NVIDIA G80



Four Architectures

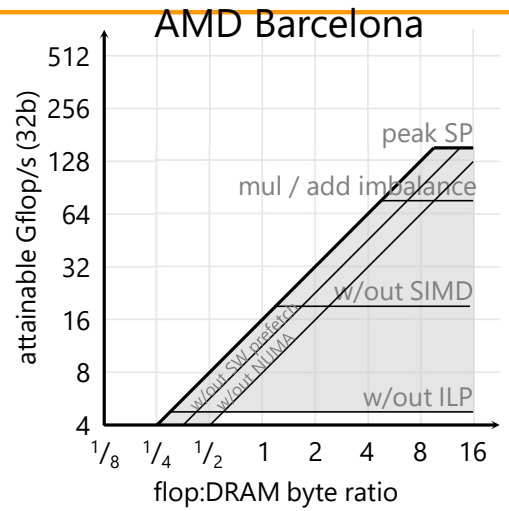


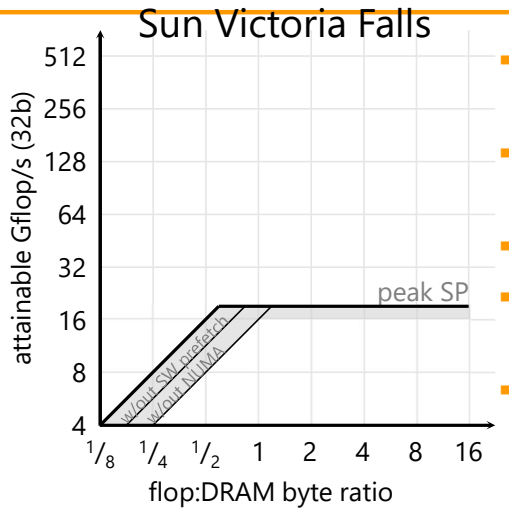
Four Architectures

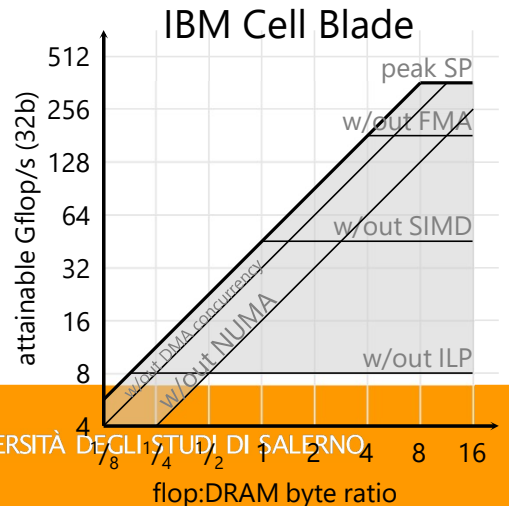


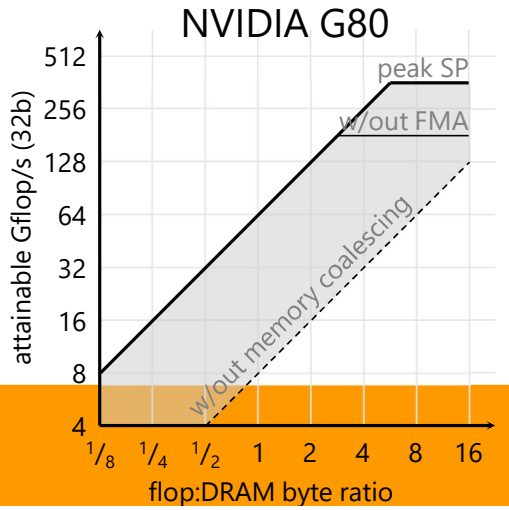
32b Rooflines for the Four

(in-core parallelism)



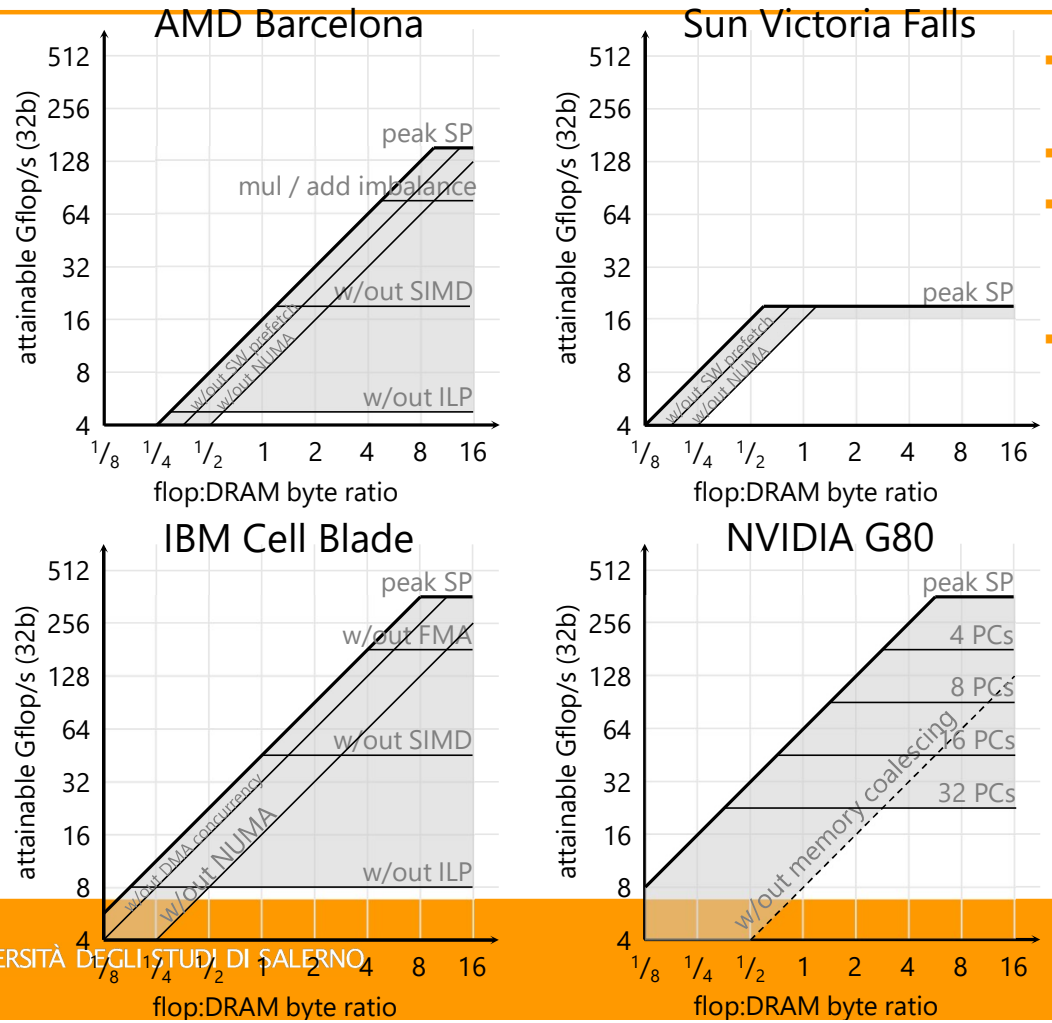






- Single Precision Roofline models for the SMPs used in this work
- Based on micro-benchmarks, experience, and manuals
- Ceilings = in-core parallelism
- Can the compiler find all this parallelism?
- NOTE:
  - log-log scale
  - Assumes perfect SPMD

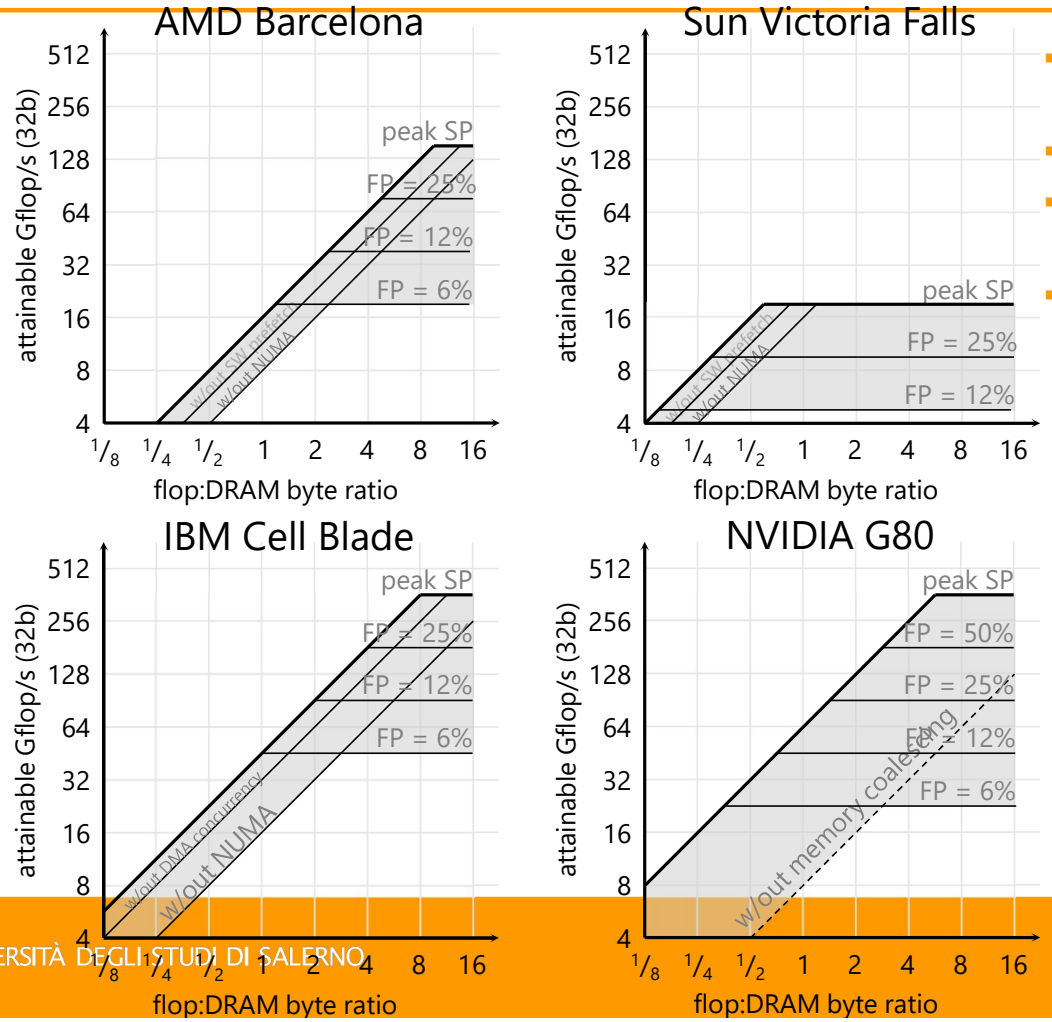
32b Rooflines for the Four
(diverged threads)



- G80 dynamically finds DLP (shared instruction fetch)
- SIMT
- If threads of a warp diverge from SIMD execution, performance is limited by instruction issue bandwidth
- Ceilings on G80 = number of unique PCs when threads diverge

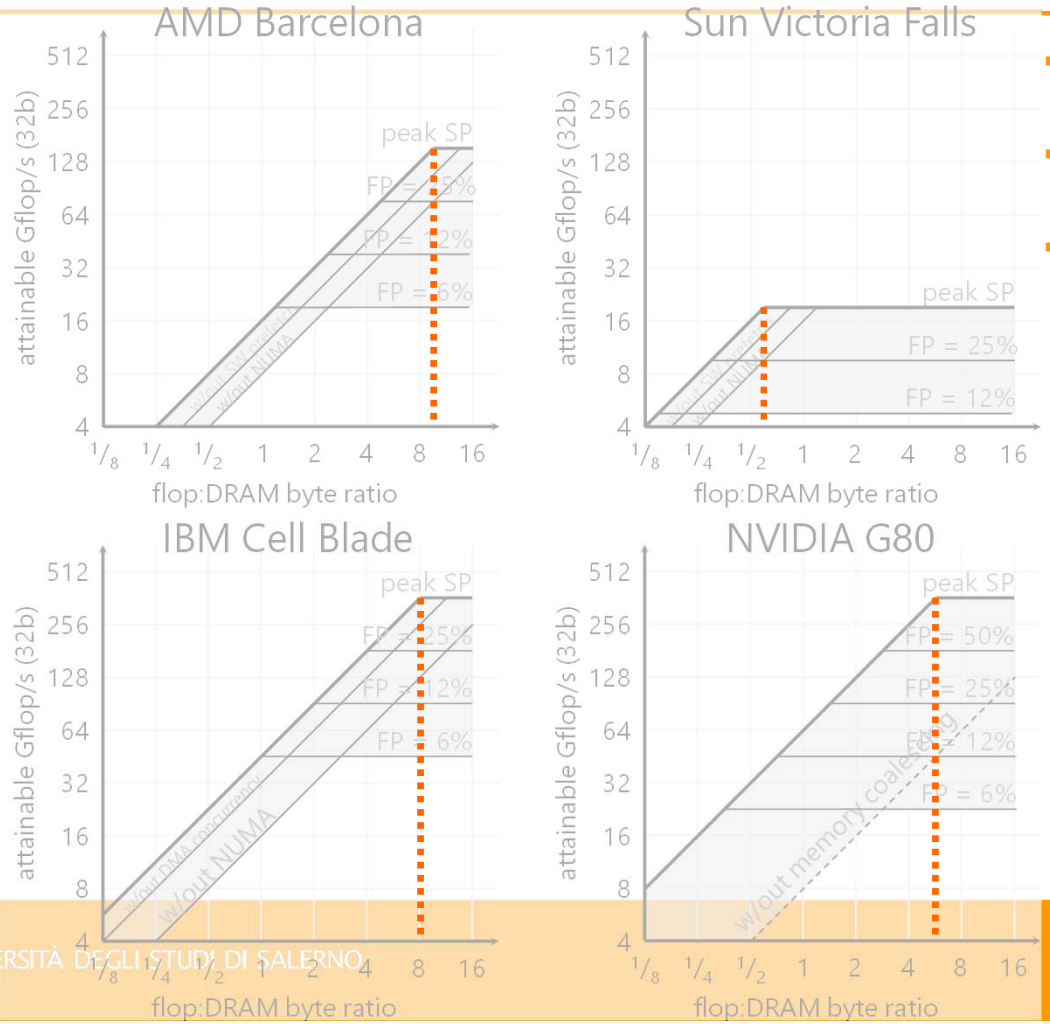
32b Rooflines for the Four

(FP fraction of dynamic instructions)



- Some kernels have large numbers of non FP instructions
- Saps instruction issue bandwidth
- Ceilings = FP fraction of dynamic instruction mix
- NOTE:
  - Assumes perfect in-core parallelism

32b Rooflines for the Four
(ridge point)



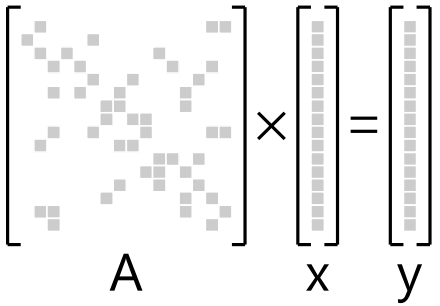
- Some architectures have drastically different ridge points
- VF may be compute bound on many kernels
- Clovertown has $^1/_3$ the BW of Barcelona = ridge point to the right

Roofline Application

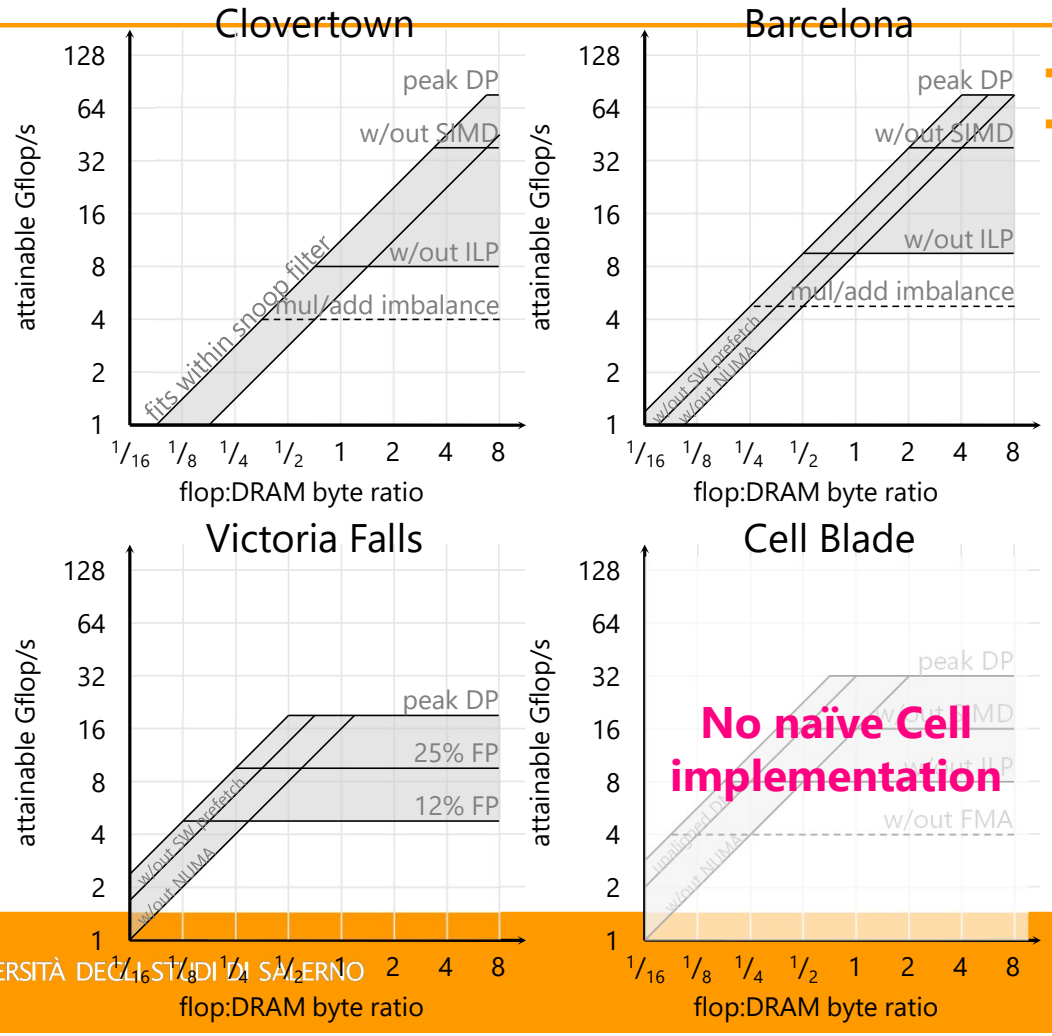
- Example kernel: Sparse Matrix-Vector Multiplication (SpMV)
  - Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, James Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms", Supercomputing (SC), 2007.

Sparse Matrix Vector Multiplication

- Sparse Matrix
  - Most entries are 0.0
  - Performance advantage in only storing/operating on the nonzeros
  - Requires significant meta data
- Evaluate y=Ax
  - A is a sparse matrix
  - x & y are dense vectors
- Challenges
  - Difficult to exploit ILP(bad for superscalar),
  - Difficult to exploit DLP(bad for SIMD)
  - Irregular memory access to source vector
  - Difficult to load balance
  - Very low arithmetic intensity (often <0.166 flops/byte) = likely memory bound

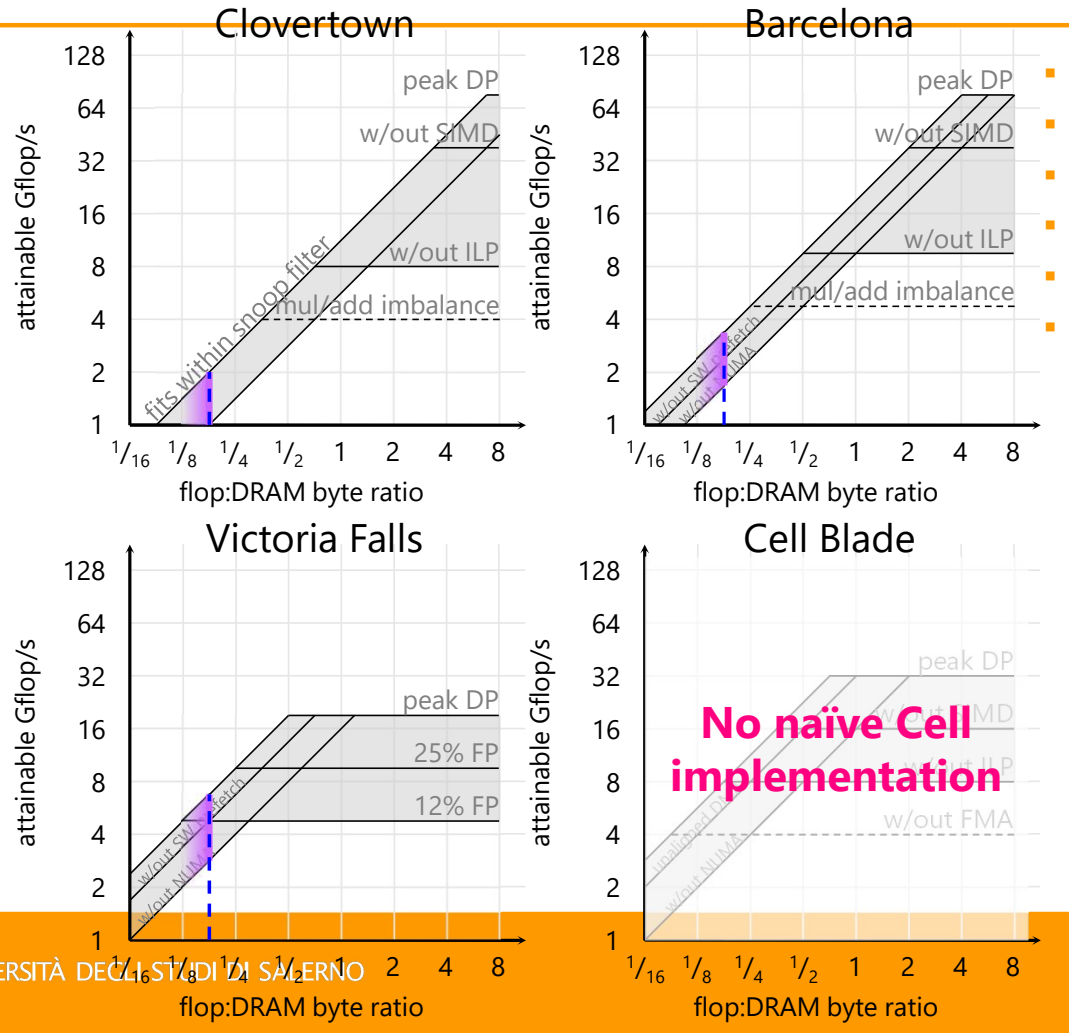


Roofline model for SpMV



- Double precision roofline models
- FMA is inherent in SpMV (place at bottom)

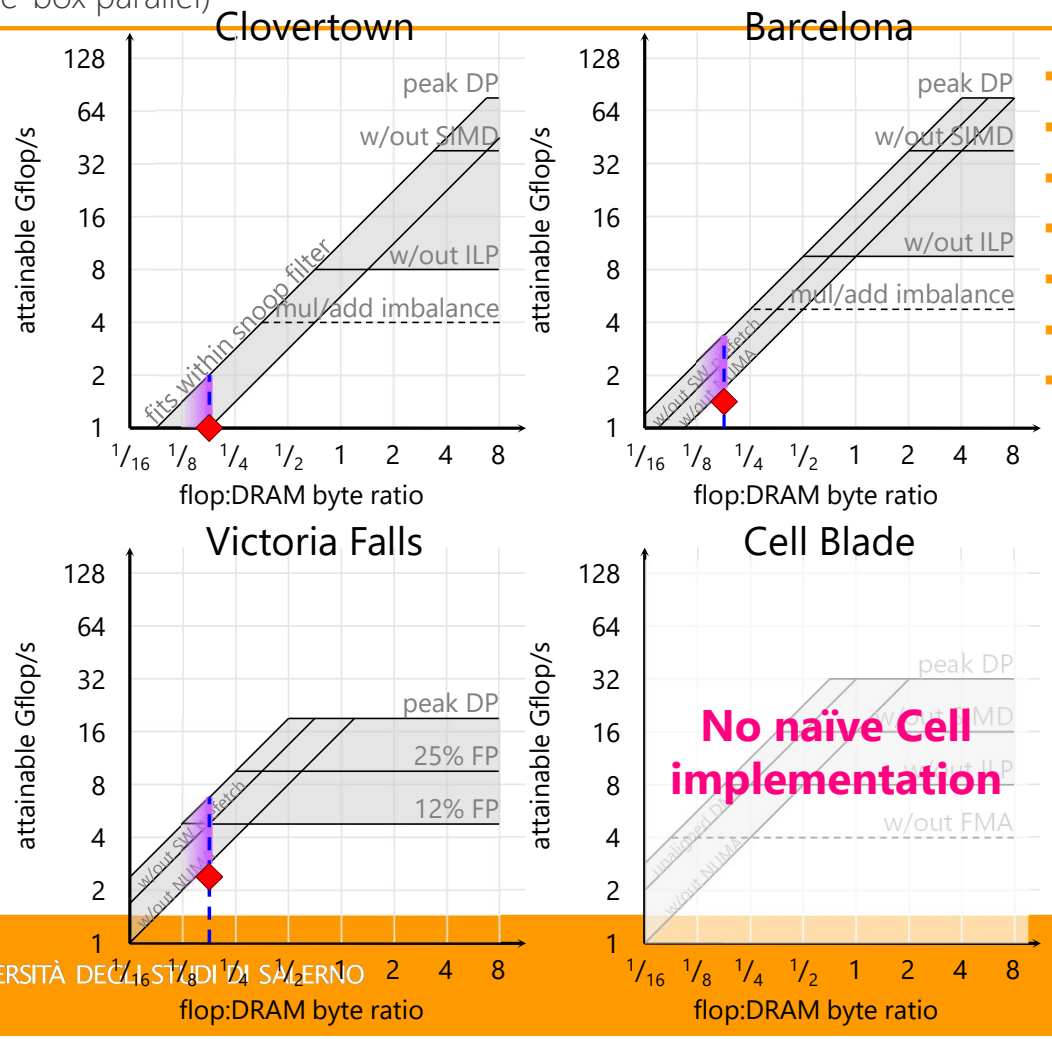
Roofline model for SpMV



- Two unit stride streams
- Inherent FMA
- No ILP
- No DLP
- FP is 12-25%
- Naïve compulsory flop:byte < 0.166

Roofline model for SpMV

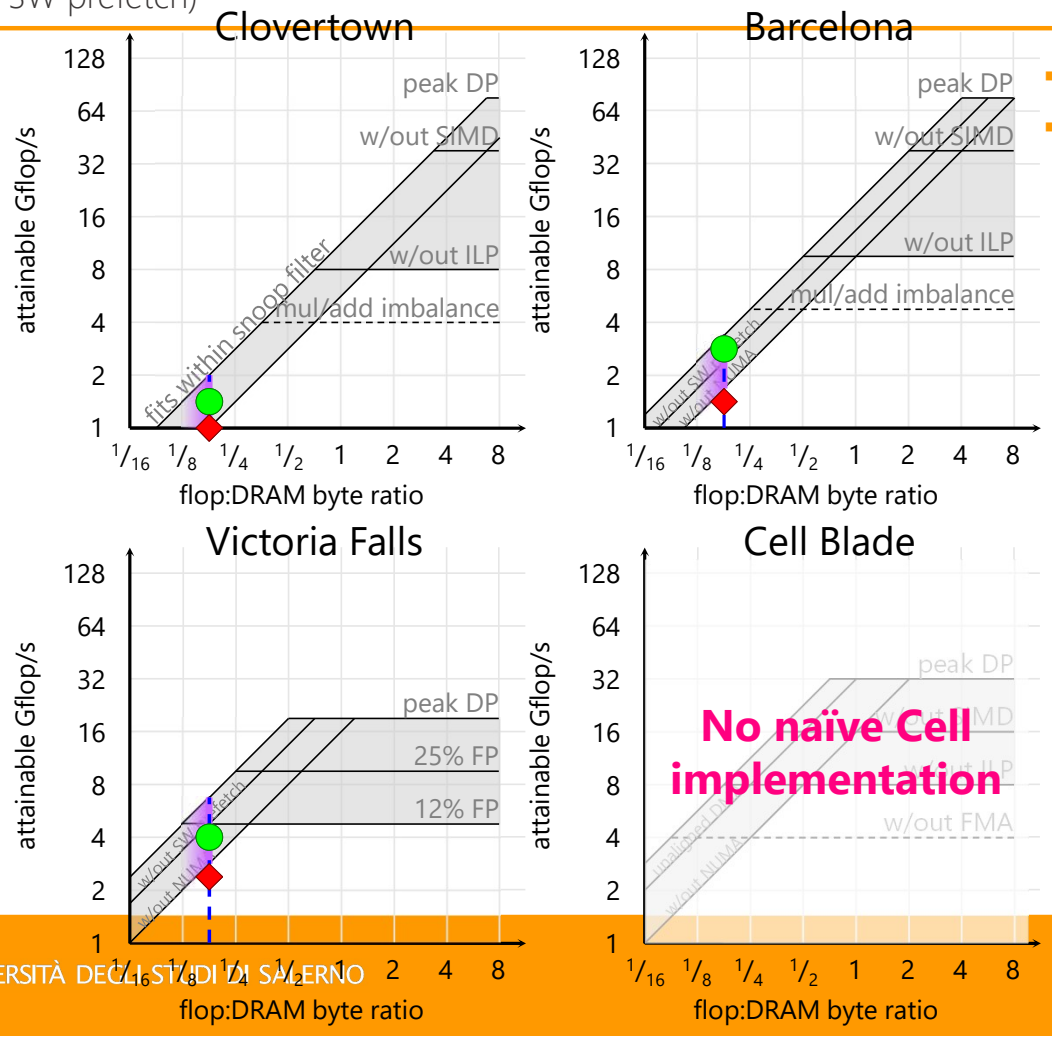
(out-of-the-box parallel)



- Two unit stride streams
- Inherent FMA
- No ILP
- No DLP
- FP is 12-25%
- Naïve compulsory flop:byte < 0.166
- For simplicity: dense matrix in sparse format

Roofline model for SpMV

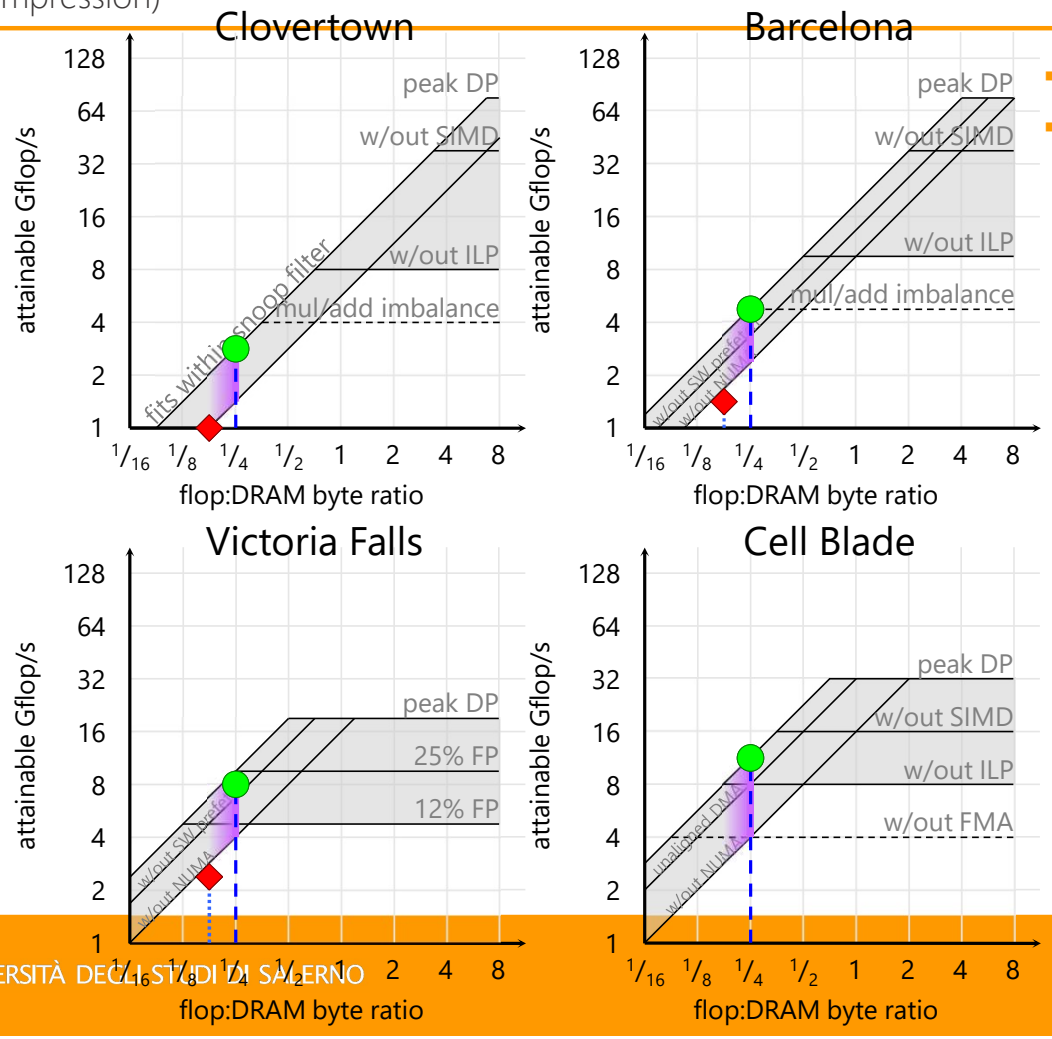
(NUMA & SW prefetch)



- compulsory flop:byte ~ 0.166
- utilize all memory channels

Roofline model for SpMV

(matrix compression)



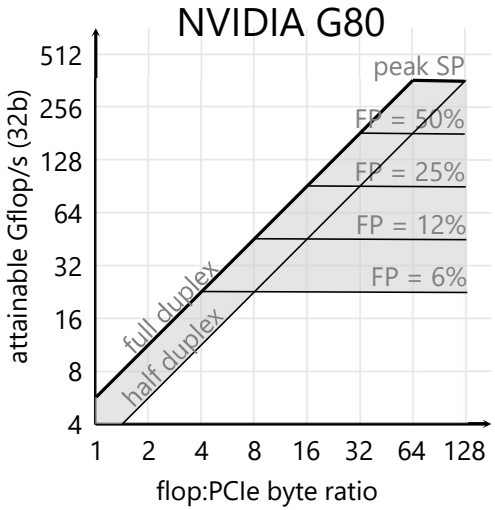
- Inherent FMA
- Register blocking improves ILP, DLP, flop:byte ratio, and FP% of instructions

Refining the Roofline: Performance Metric

- There is no reason either floating point (Gflop/s) must be the performance metric
- Could also use:
  - Graphics (Pixels, Vertices, Textures)
  - Crypto
  - Integer
  - Bitwise
  - etc…

Refining the Roofline: Bandwidth

- For our kernels, DRAM bandwidth is the key communication component
- For other kernels, other bandwidths might be more appropriate
  - L2 bandwidth (e.g., DGEMM)
  - PCIe bandwidth (offload to GPU)
  - Network bandwidth
- Example: zero overhead double buffered transfers to/from a GPU over PCIe x16
  - How bad is a SP stencil ?
  - What about SGEMM ?
- No overlap / high overhead tends to smooth performance
  - Performance is half at ridge point



Mix and match

- In general, you can mix and match as the kernel/architecture requires:
- e.g. all possibilities is the cross product of performance metrics with bandwidths

{Gflop/s, GIPS, crypto, …} × {L2, DRAM, PCIe, Network}

Lab Exercises

1. Visualize the roofline model for the `matmul` codes developed in previous lectures
   - calculate max bandwidth
   - compare different implementations
     - with/without SIMD
     - with/without multithreading
     - optionally NUMA optimizations
2. Visualize the roofline model for code with different arithmetic intensity