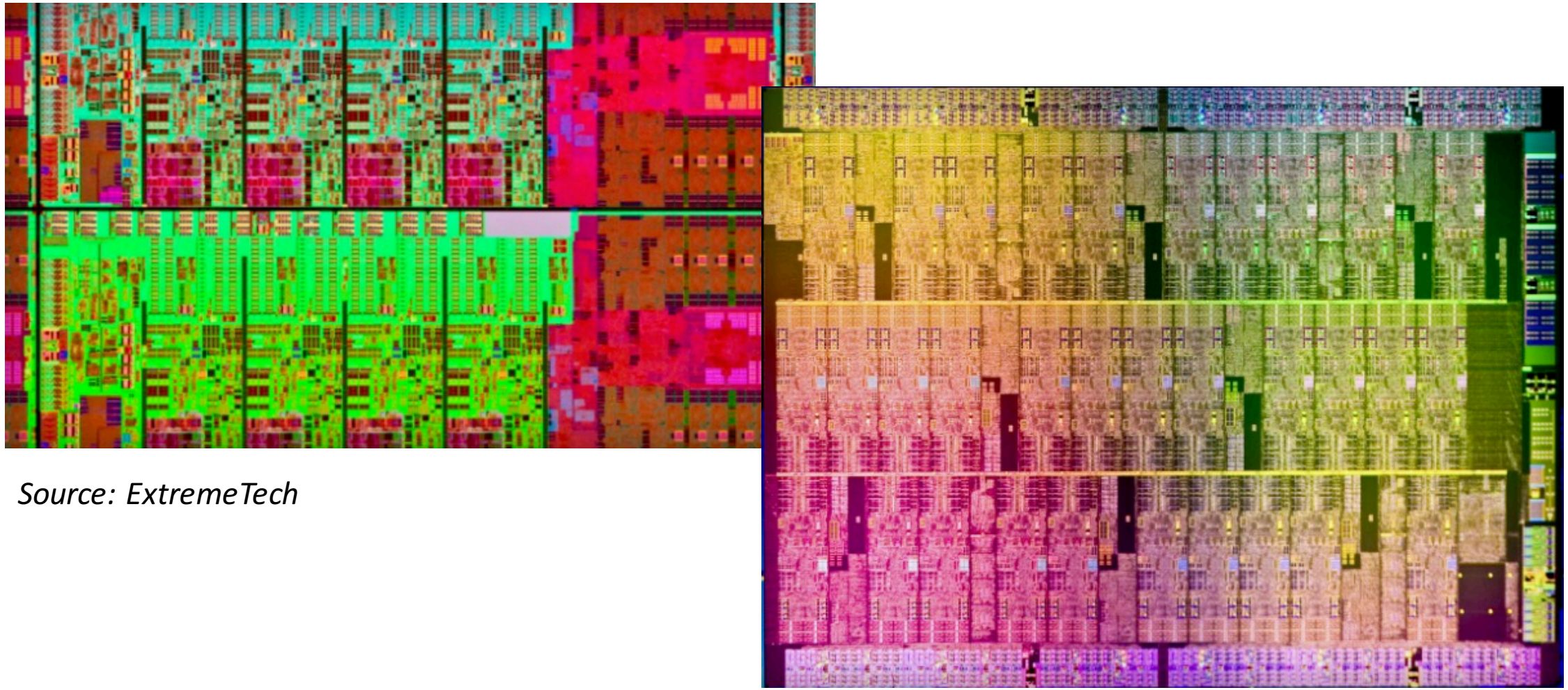
Squeezing The Hardware To Make Performance Juice

Sasha Goldshtein @goldshtn
CTO, Sela Group github.com/goldshtn

Agenda

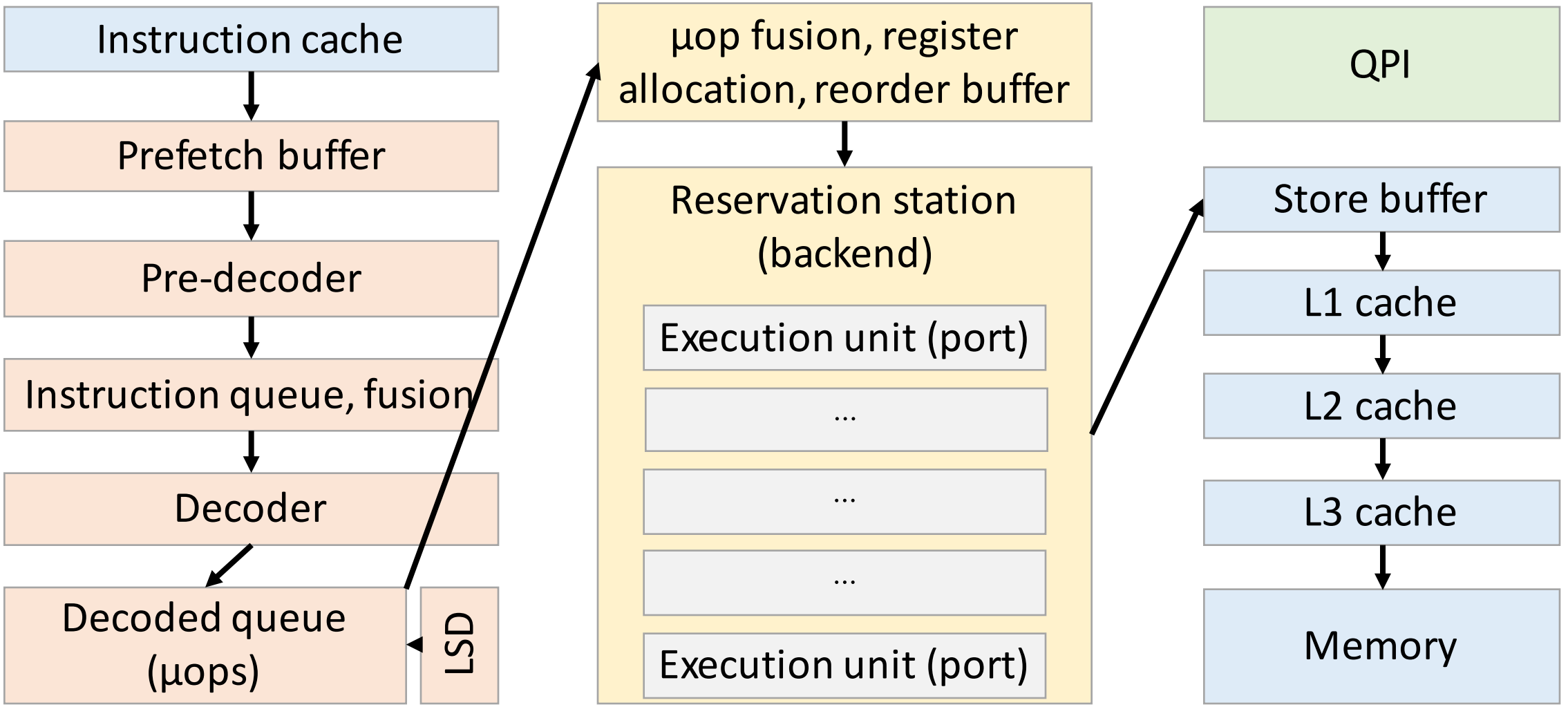
- Modern CPU organization
- Demonstrations of fundamental CPU architecture
- Deep-dive on vectorization
- Understanding the cache
- Scaling bottlenecks
- The future?

A Modern Processor



Source: ExtremeTech

Processor Organization: Intel Nehalem+



Skylake Backend (sustained ≈4 µops/cycle)

Port 0

- branch
- int/vector arithmetic
- vector string
- FP add, mul, FMA, AES
- integer vector mul

Port 1

- int/vector arithmetic
- int mul, bit scan
- FP add, mul, FMA, AES
- integer vector mul

Port 6

- int arithmetic
- jmp, branch

Port 2 – load

Port 3 – load

Port 4 – store

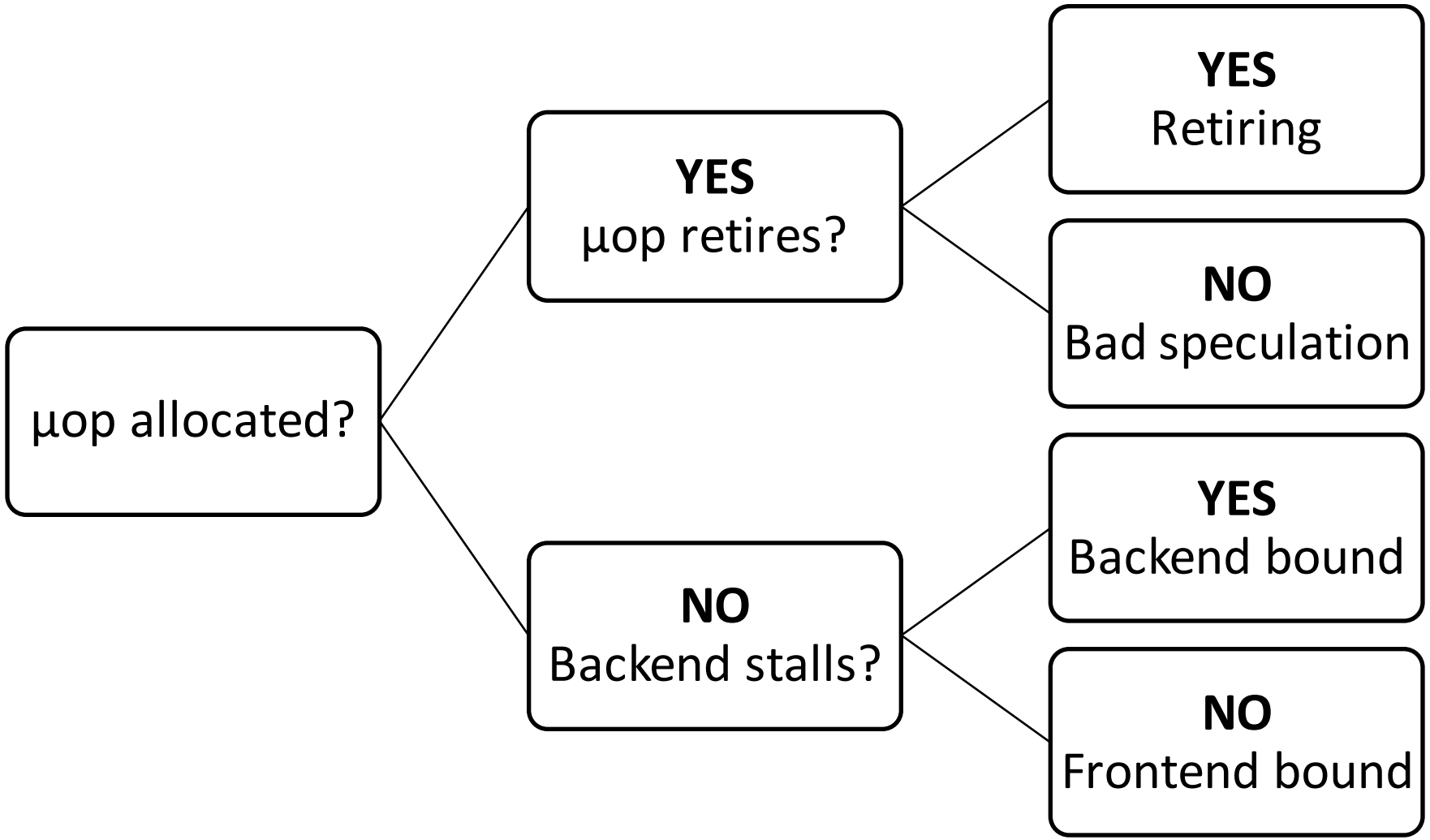
Port 7 – addr gen

Port 5

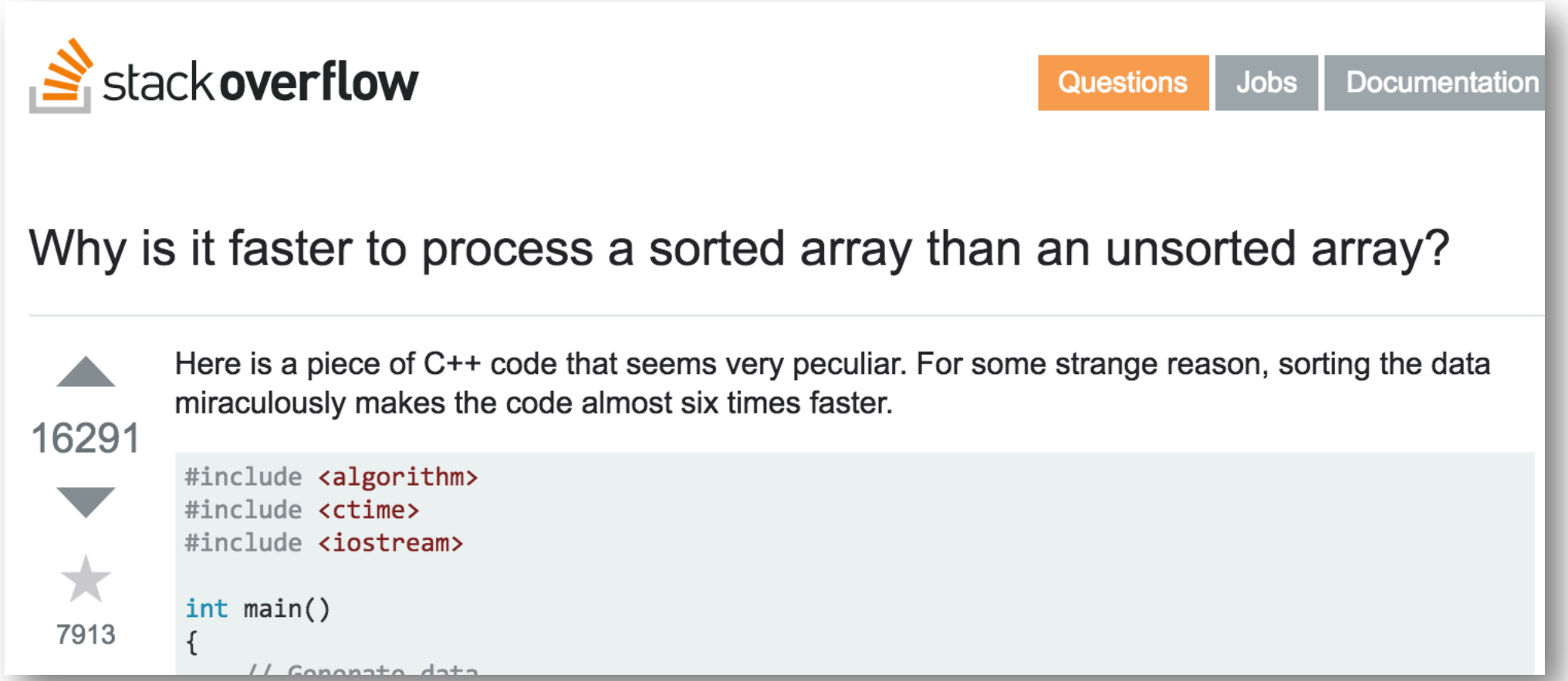
- int/vec arith
- vec permute
- FPadd, PCLMUL

- Almost all vector units are 256-bit (32B)
- Max 64BR and 32BW/cycle (4 cycle latency to L1)

Intel Top-Down Approach



Processor Behavior: Branch Prediction



Processor Behavior: Branch Prediction

```
int sum = 0;
foreach (int i in _sorted)
  if (i >= 128)
    sum += i;

int sum = 0;
foreach (int i in _unsorted)
  if (i >= 128)
    sum += I;
```

Method	Mean Time (ns)
Sorted	18.2571
Unsorted	121.7550

Processor Behavior: ILP

```
int min = …, max = …;
for (int i = 0; i < data.Length; ++i)
{
  min = Math.Min(min, data[i]);
  max = Math.Max(max, data[i]);
}
```

Method	Mean Time
Naive	498.6605
Optimized	682.0700

```
int min1, min2, max1, max2 = …;
for (int i = 0; i < data.Length; i += 2)
{
  min1 = Math.Min(min1, data[i]); min2 = Math.Min(min2, data[i+1]);
  max1 = Math.Max(max1, data[i]); max2 = Math.Max(max2, data[i+1]);
}
int min = Math.Min(min1, min2), max = Math.Max(max1, max2);
```

“Optimized”

Naive

```
LOOP:  movsxd  r8,eax
mov     r8d,dword ptr [rcx+r8*4+10h]
cmp     edi,r8d
jle     MIN
mov     edi,r8d
jmp     MIN
MIN:   cmp     esi,r8d
jge     MAX
jmp     NOMAX
MAX:   mov     r8d,esi
NOMAX: mov     esi,r8d
inc     eax
cmp     edx,eax
jg      LOOP
```

Optimized

```
test    r11d,r11d
jle     DONE
mov     rsi,rcx
cmp     r10d,r11d
jae     RANGE_FAIL
movsxd  rdi,r10d
mov     esi,[rsi+rdi*4+10h]
mov     rdi,rcx
lea     ebx,[r10+1]
cmp     ebx,r11d
jae     RANGE_FAIL
movsxd  rbx,ebx
mov     edi,[rdi+rbx*4+10h]
cmp     r8d,esi
jle     00007ffe`8c8b066d
mov     r8d,esi
jmp     00007ffe`8c8b066d
```

```
cmp     r9d,edi
jle     00007ffe`8c8b0677
mov     r9d,edi
jmp     00007ffe`8c8b0677
cmp     eax,esi
jge     00007ffe`8c8b067d
jmp     00007ffe`8c8b067f
mov     esi,eax
mov     eax,esi
cmp     edx,edi
jge     00007ffe`8c8b0687
jmp     00007ffe`8c8b0689
mov     edi,edx
mov     edx,edi
add     r10d,2
cmp     r11d,r10d
jg      LOOP
```

Processor Behavior: Vector Units

```
int min = …, max = …;
for (int i = 0; i < data.Length; ++i)
{
  min = Math.Min(min, data[i]);
  max = Math.Max(max, data[i]);
}
```

Method	Mean Time
Naive	512.8992
SIMD	105.4331

```
Vector<int> vmin = …, vmax = …;
for (int i = 0; i < data.Length; i += Vector<int>.Count)
{
  Vector<int> v = new Vector<int>(data, i);
  var mask1 = Vector.LessThan(v, vmin), mask2 = Vector.GreaterThan(v, vmax);
  vmin = Vector.ConditionalSelect(mask1, v, vmin);
  vmax = Vector.ConditionalSelect(mask2, v, vmax);
}
// Find min and max from vectors
```

Processor Behavior: Parallelization

```
int min = …, max = …;
for (int i = 0; i < data.Length; ++i)
{
  min = Math.Min(min, data[i]);
  max = Math.Max(max, data[i]);
}
```

Method	Mean Time
Naive	491.5137
ILP	665.0614
SIMD	108.2504
Parallelized	193.9453

```
int[] mins = …, maxs = …;
Parallel.For(0, Environment.ProcessorCount, i => {
  …
  for (int j = from; j < to; ++j) { … }
});
// Find min and max from arrays
```

Vector Instructions

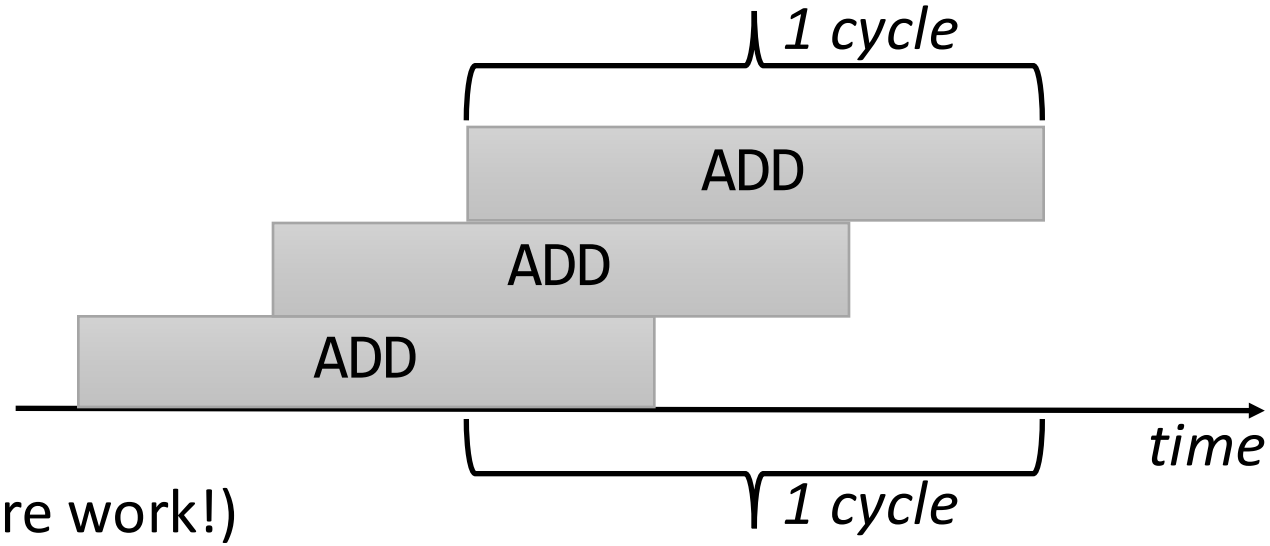
- Classic instructions operate on scalar values
  - `ADD  ECX,                DWORD PTR [EAX]`
  - `MOV  QWORD PTR [RBX], R9`
  - `FMUL DWORD PTR [EAX]`
- SIMD = Single Instruction Multiple Data
- Operand width ranging from 128-bit (SSE2+) to 512-bit (AVX-512)
  - `ADDPS  XMM0, XMMWORD PTR [EAX]`
  - `MOV    XMM1, XMMWORD PTR [ECX]`
  - `VMULPS YMM2, YMM1, YMM0`

Categories of Vector Instructions

- Arithmetic (add, mul, mask, horizontal add)
- Compare
- Conversion (integer to floating-point, rounding)
- Cryptography (AES, CRC, SHA)
- Math (e^x, log, $1/\sqrt{}$, trigonometry, min, max, truncate, CDF, erf)
- Memory (load/store, scatter/gather, broadcast)
- String compare

SIMD Performance

- Latency (start to finish):
  - ADD – 1 cycle
  - PADDD – 1 cycle
- Throughput (sustained):
  - ADD – 3 per cycle
  - PADDD – 2 per cycle (but 4× more work!)



Vectorization: Matrix Multiplication

```
for (int i = 0; i < …; ++i)
  for (int k = 0; k < …; ++k)
    for (int j = 0; j < …; ++j)
      c[i*… + j] += a[i*… + k] * b[k*… + j];
```

```
for (int i = 0; i < …; ++i)
  for (int k = 0; k < …; ++k) {
    Vector<float> va = new Vector<float>(a[i*… + k]);
    for (int j = 0; j < …; j += Vector<float>.Count) {
      Vector<float> vb = new Vector<float>(b, k*… + j);
      Vector<float> vc = new Vector<float>(c, i*… + j);
      vc += va * vb;
      vc.CopyTo(c, i*… + j);
    }
  }
```

Method	Mean Time
Scalar	217.7963
SIMD	43.5546

Sidebar: V4FMADDPS

- Apparently, we should expect a `V4FMADDPS` instruction in an upcoming processor
- `V4FMADDPS ZMM0, ZMM1-4, addr`
  1. t1 = read 32-bits from addr
  2. z1 = (t1, t1, …, t1)
  3. ZMM0 = ZMM0 + ZMM1*z1
  4. t2 = read 32-bits from addr+4
  5. z2 = (t2, t2, …, t2)
  6. ZMM0 = ZMM0 + ZMM2*z2
  7. …
  8. ZMM0 = ZMM0 + ZMM4*z4

Sidebar: Vectorization ≠ Parallelization

- Suppose we're adding 32-bit integers…
- On Skylake i7 (e.g. i7-6700HQ, $378 RCP):
  - Three integer vector arithmetic units
  - 256-bit instructions, operating on pairs of eight-`int` vectors
  - Four cores
- 3×8×4=96 potential speedup compared to scalar single-core

Vectorization: Vector Normalization

- Normalizing a 3D vector (point) across its components:

```
for (int i = 0; i < pts.Length; ++i)
{
  Point3 pt = pts[i];
  float norm = (float)Math.Sqrt(pt.X*pt.X + pt.Y*pt.Y + pt.Z*pt.Z);
  pt.X /= norm;
  pt.Y /= norm;
  pt.Z /= norm;
  pts[i] = pt;
}
```

- How to vectorize?

Vectorization: AoS vs. SoA

- We need `pt.X`, `pt.Y`, `pt.Z` to be SIMD vectors, then `norm` is a SIMD vector and all is well
- Reorganize data as Structure of Arrays

Method	Mean Time
Scalar	34.6896
SIMD	8.3120

```
for (int i = 0; i < xs.Length; i += vecSize)
{
  Vector<float> x = new Vector<float>(xs, i),
                y = new Vector<float>(ys, i),
                z = new Vector<float>(zs, i);
  Vector<float> norm = Vector.SquareRoot(x * x + y * y + z * z);
  x /= norm; y /= norm; z /= norm;
  x.CopyTo(xs, i);
  y.CopyTo(ys, i);
  z.CopyTo(zs, i);
}
```

Vectorization: N-Body Simulation

- Particles have a mass and a 3D position, velocity, and acceleration
- Simulate gravity interactions between each pair of particles:

```
Vector3 diff = position - other.Position;
float r2 = (diff * diff).Sum();
diff = diff * (1.0f / ((float)Math.Sqrt(r2) * (r2 * 1.0f)));
Acceleration = Acceleration - diff * other.Mass;
other.Acceleration = other.Acceleration + diff * Mass;
```

- Now update position and velocity according to the new values and the elapsed simulation time

Vectorization: AoS vs. SoA

- Vectorizing this seems difficult because SIMD “packets” of related data are not organized consecutively (Array of Structures)

P1.X	P1.Y	P1.Z	P1.VX	P1.VY	P1.VZ	P1.AX	P1.AY	P1.AZ	P2.X	P2.Y	P2.Z	P2.VX	P2.VY

  - Consider what it would mean to calculate `X - other.X` over two SIMD “packets” of particles
- Reorganize data as Structure of Arrays:
  - Apply forces from another packet to each of my particles
  - Then apply “inside forces”

P1.X	P2.X	P3.X	P4.X	P5.X	P6.X	P7.X	P8.X
P1.Y	P2.Y	P3.Y	P4.Y	P5.Y	P6.Y	P7.Y	P8.Y
P1.Z	P2.Z	P3.Z	P4.Z	P5.Z	P6.Z	P7.Z	P8.Z
P1.VX	P2.VX	P3.VX	P4.VX	P5.VX	P6.VX	P7.VX	P8.VX
…	…	…	…	…	…	…	…

Vectorization: `strstr`

```
// Simple, no-frills approach:
return haystack.Contains(needle);

// Vectorized version:
__m256i first = _mm256_set1_epi8(needle[0]);
__m256i last = _mm256_set1_epi8(needle[ns-1]);
for (int i = 0; i < hs; i += 32) {
  __m256i block_first = _mm256_loadu_si256(&haystack + i);
  __m256i block_last  = _mm256_loadu_si256(&haystack + i + ns - 1);
  __m256i eq_first    = _mm256_cmpeq_epi8(first, block_first);
  __m256i eq_last     = _mm256_cmpeq_epi8(last, block_last);
  unsigned mask = _mm256_movemask_epi8(_mm256_and_si256(eq_first, eq_last));
  while (mask != 0) { … memcmp(haystack + i + first_bit_set_pos + 1, … }
}
```

Method	Mean Time
Scalar	1,872.2798
SIMD	202.1229

Vectorization: Sorted Set Intersection

```
int i = 0, j = 0;
int outCounter  = 0;
while (i < _set1.Length  && j < _set2.Length)
{
  if (_set1[i] < _set2[j])
    i++;
  else if (_set2[j] < _set1[i])
    j++;
  else
  {
    _output[outCounter++] = _set1[i];
    i++; j++;
  }
}
```

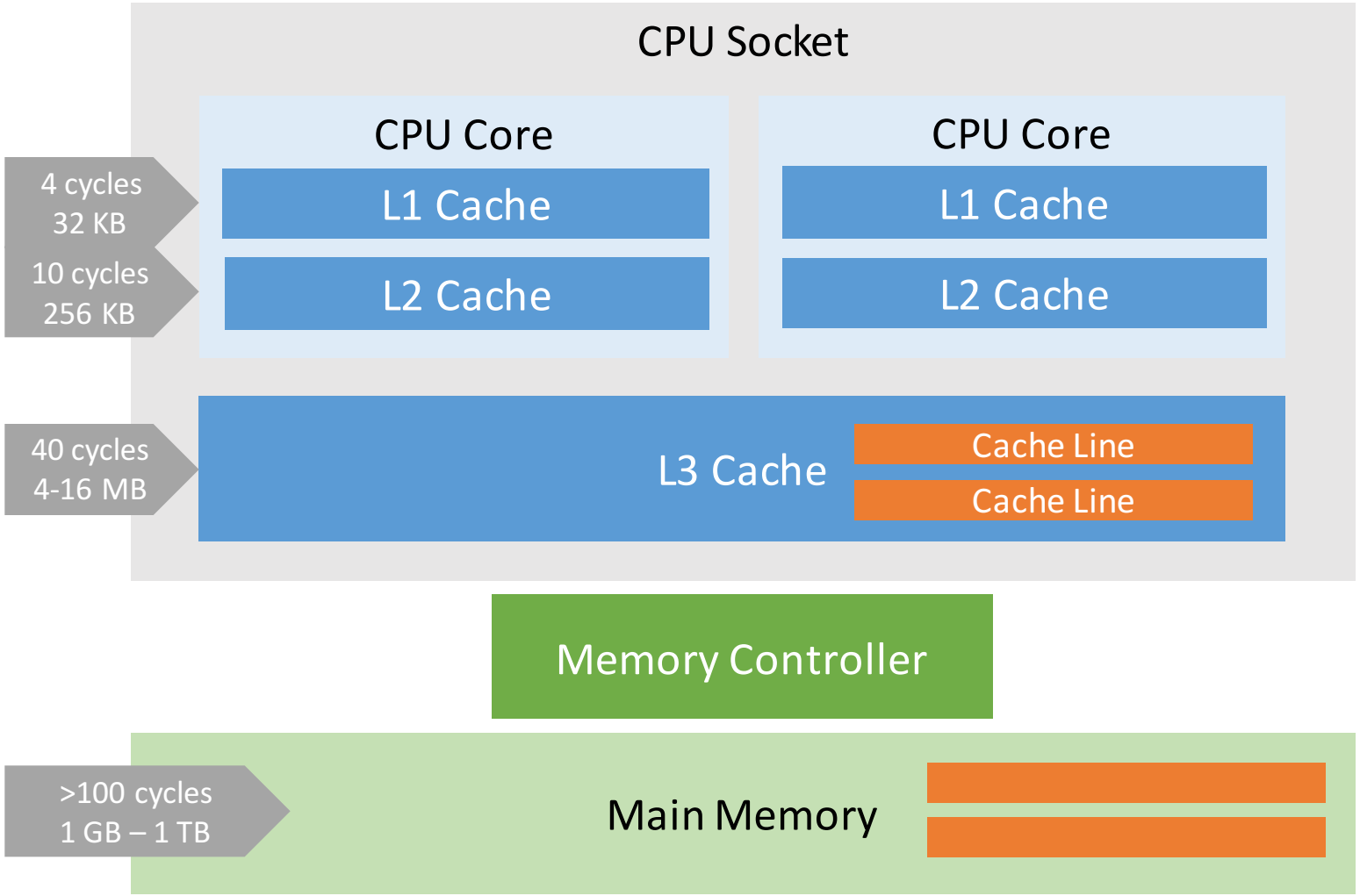
1	3	11	13	18	28
-1	2	3	18	27	28

Vectorization: Sorted Set Intersection

```
size_t count = 0;
size_t i_a = 0, i_b = 0;
while(i_a < size && i_b < size) {
  __m128i v_a = _mm_loadu_si128((__m128i*)&A[i_a]);
  __m128i v_b = _mm_loadu_si128((__m128i*)&B[i_b]);
  __m128i res_v = _mm_cmpestrm(v_b, 8, v_a, 8,
                _SIDD_UWORD_OPS|_SIDD_CMP_EQUAL_ANY|_SIDD_BIT_MASK);
  int r = _mm_extract_epi32(res_v, 0);
  __m128i p = _mm_shuffle_epi8(v_a, shuffle_mask[r]);
  _mm_storeu_si128((__m128i*)&C[count], p);
  count += _mm_popcnt_u32(r);
  short a_max = _mm_extract_epi16(v_a, 7);
  short b_max = _mm_extract_epi16(v_b, 7);
  i_a += (a_max <= b_max) * 4;
  i_b += (a_max >= b_max) * 4;
}
```

Method	Mean Time
Scalar	36,908.9380
SIMD	9,779.6098

Cache Structure, typical i5 (no L4 EDRAM)



Cache: Data Access Reordering

```
// a, b, c are 512×512 matrices of floats
// total size of all three: 3MB

for (int i = 0; i < …; ++i)
  for (int j = 0; j < …; ++j)
    for (int k = 0; k < …; ++k)
      c[i*… + j] += a[i*… + k] * b[k*… + j];

for (int i = 0; i < …; ++i)
  for (int k = 0; k < …; ++k)
    for (int j = 0; j < …; ++j)
      c[i*… + j] += a[i*… + k] * b[k*… + j];
```

Method	Mean Time
Naive	378.8371
Reorg	217.7963

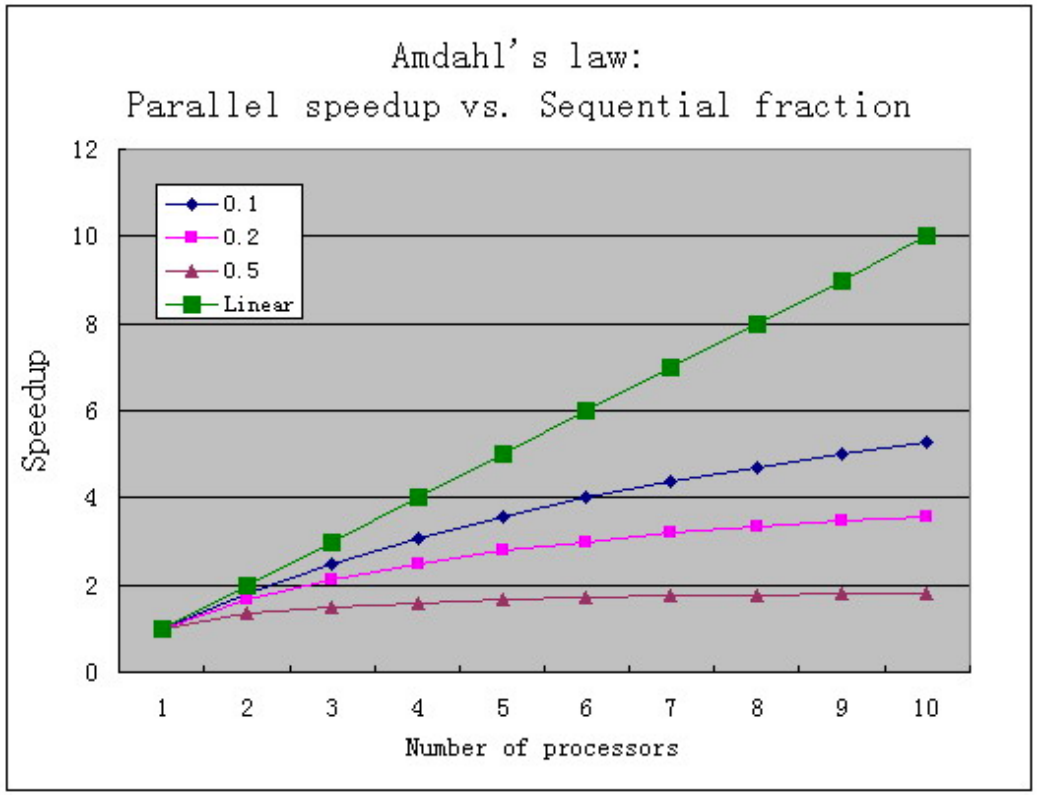
Cache: Tiling

```
// image, rotated are 1024×1024 int[]
// total size of both: 8MB

for (int y = 0; y < ROWS; ++y)
  for (int x = 0; x < COLS; ++x)
  {
    rotated[x*ROWS  + y] = image[y*COLS  + x];
  }

for (int y = 0; y < ROWS; y += BH) for (int x = 0; x < COLS; x += BW)
  for (int by = 0; by < BH; ++by)  for (int bx = 0; bx < BW; ++bx)
  {
    rotated[(x+bx)*ROWS  + (y+by)] = image[(y+by)*COLS  + (x+bx)];
  }
```

Method	Mean Time
Naive	4.3124
Blocked	2.3054

Amdahl's Law

- 8, 16, 24, 44, 72 cores make locking uncomfortable and impractical
- Seek lock-free solutions, message passing, avoid shared state
- Note that `Monitor.Enter`-free isn't lock-free



Source: Wikimedia Commons, CC-BY-SA 3.0

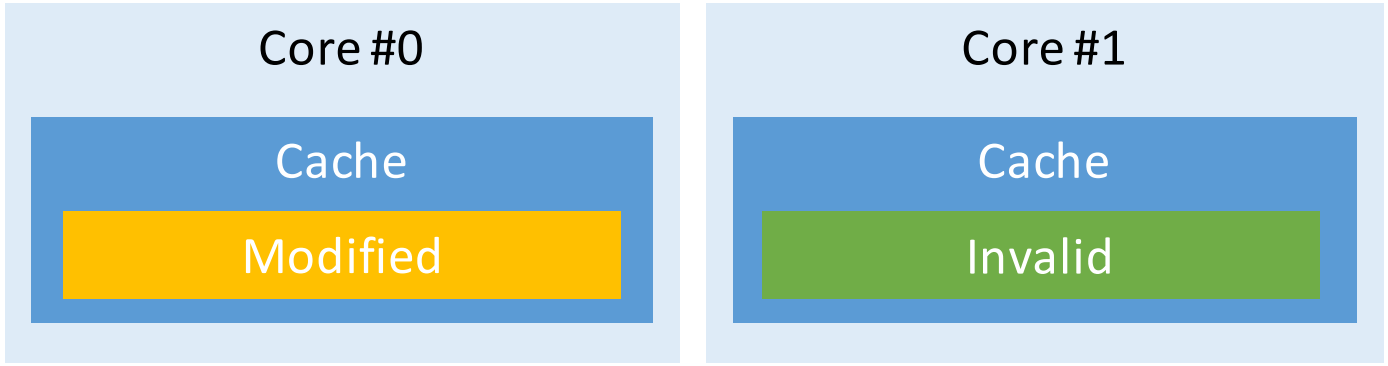
Memory Bottleneck

- Typical desktop processors have ≈25GB/s memory bandwidth
  - Each core can issue 2×32-byte reads from L1 cache each cycle, translates to >>100GB/s but the memory system is considerably slower
- A lot of algorithms easily hit the memory wall on 4-8 cores or less
- Single-core dot product (max loads on this processor: ≈25GB/s)

Method	Time/iteration (us)	GB loads/sec	Cycles/vector
DP scalar	975.574	8.008	5.199
dpps	430.409	18.151	2.294
muladd	335.575	23.281	1.788
FMA	340.388	22.952	1.814

MESI Protocol and Cache Invalidation

- The MESI protocol guarantees cache coherence
- MESI stands for line states: Modified, Exclusive, Shared, Invalid
- Complex inter-CPU messaging guarantees correct state transitions



Cache Invalidation

```
double[] partialIntegrals = new double[P];

// Create P threads, each runs:
for (int i = from; i < to; ++i)
  partialIntegrals[threadIndex] += f(…);

// Create P threads, each runs:
double myPart = 0.0;
for (int i = from; i < to; ++i)
  myPart += f(…);
partialIntegrals[threadIndex] = myPart;
```

Method	# Threads	Mean Time
Sequential	1	54.3311
Parallel1	1	54.8744
Parallel2	1	54.7636
Parallel1	2	52.9750
Parallel2	2	28.1005
Parallel1	4	54.8956
Parallel2	4	15.8295

Tools Are Available

- Modern processors have a PMU with PMCs
  - LLC misses, branch mispredictions, instructions retired, µops decoded, etc.
  - Fire interrupt after a PMC was incremented N times
- Intel Parallel Studio has tools for correlating PMU events with code and issuing guidance
  - Intel VTune Amplifier, Intel Threading Advisor, Intel Vector Advisor
- Windows can track PMU events through ETW – a little raw
  - Proof-of-concept parser of `tracelog -PMC` recording

“The Future” – Speculation!

- Cheap, low-energy cores alongside more expensive ones, e.g. mobile
- External processing units, such as KNC/KNL boards
- Offloading to GPU
- Offloading to FPGA

Summary

- Modern CPUs are complex, but understandable
- There are many easy wins from vectorization, parallelization, better data organization, cache friendliness
- Most of this can be done from C#, no need for C++
- Use tools when not sure

Thank You!

Sasha Goldshtein

@goldshtn