Scalability and optimizations

INF8601 – Systèmes parallèles
Automne 2015

Francis Giraldeau
francis.giraldeau@polymtl.ca

Prof. Michel Dagenais
michel.dagenais@polymtl.ca

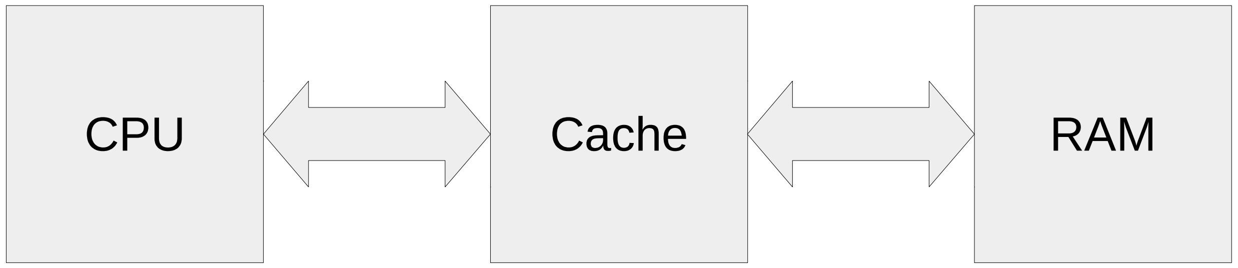
École Polytechnique de Montréal

Plan

- Roofline model
- Arithmetic intensity
- Cache optimizations

Computation chain

- The slowest component limits the performance
- On a multicore processor, memory bandwidth is shared
- If the bus is saturared, the processor will stall



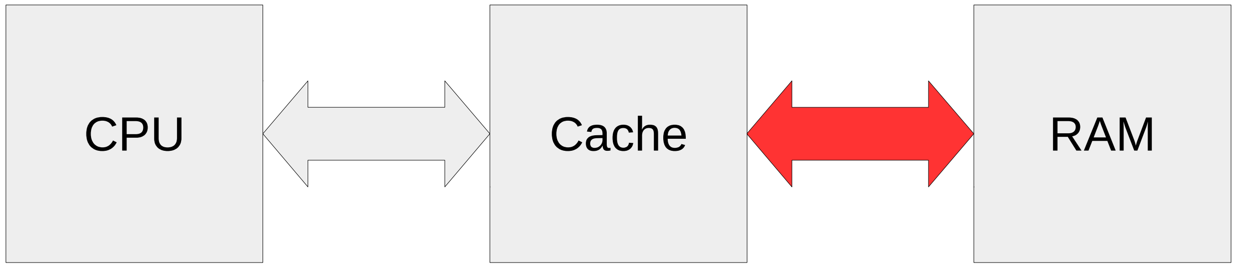
Processing components

- Arithmetic operation
  - Add, sub: 1 cycle
  - Mul: 3 cycles
  - Div: 100 cycles
  - Latency hidden by the pipeline
- Communication (load/store)
  - Traffic on the main bus for cold accesses
  - Traffic regs <-> L1 for data in the cache

Cache misses

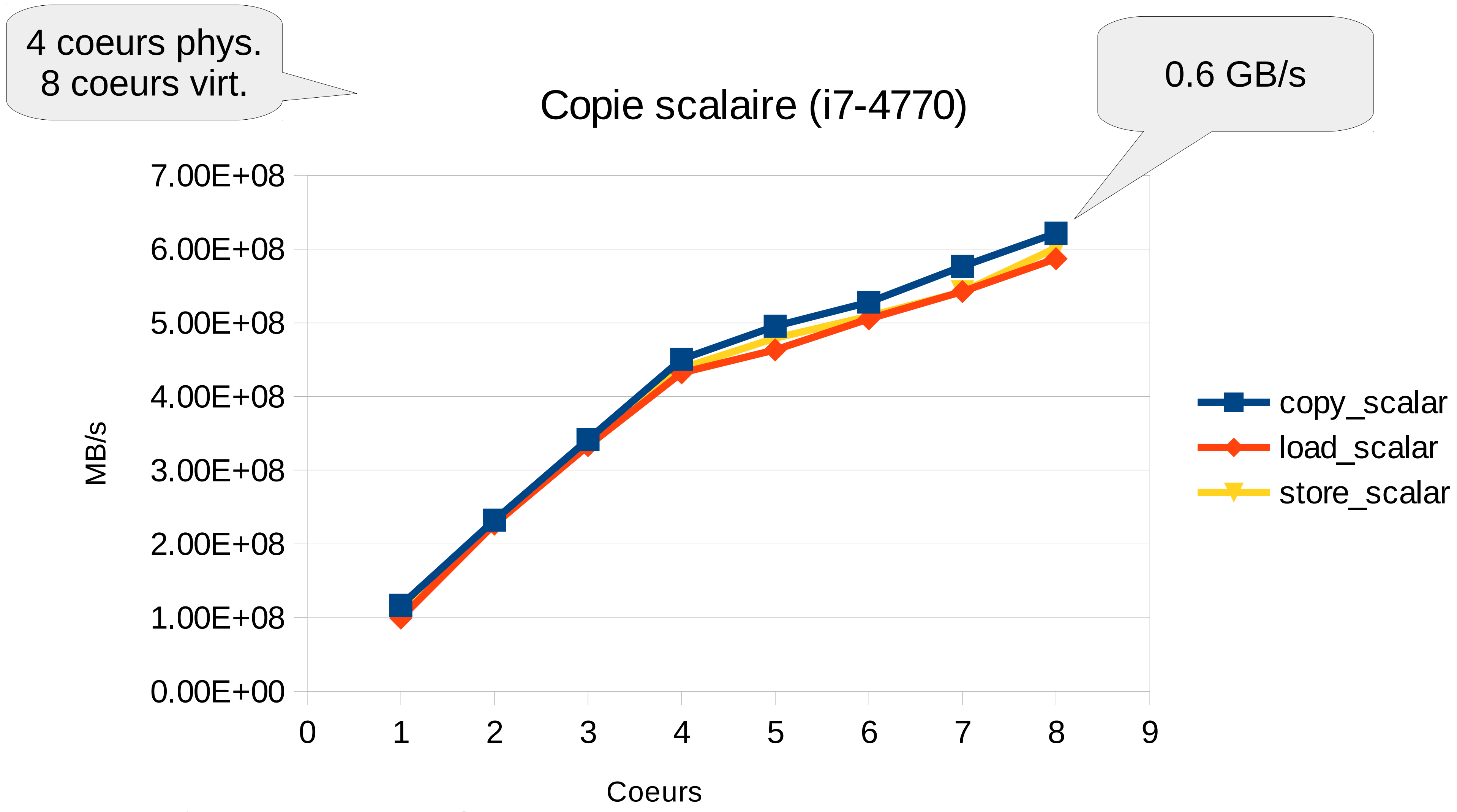
1) Cold start
   - Impossible to eliminate all
   - *Hardware prefetching*: detects regular access patters (forward and backward) and prefetch the cache lines.
   - Pipeline SuperScalaire (HyperThreading)
   - Explicite instruction PREFETCH
2) Eviction: replace old data
   - L1 cache capacity: ~32kio
   - Increase cache size ($$$)
   - Work on a smaller set of data
   - Compact the data
3) Conflict: Same cache line for two entries
   - Solution: cache associativity reduces conflics
4) Conflit: Interference (multicore)
   - Example: false sharing
   - Prevented with distinct cache lines
   - Alignment and padding to prevent interference

Study of **bandwidth** (MB/s)
Intel 8 cores HyperThread (4 phys, 8 virt)



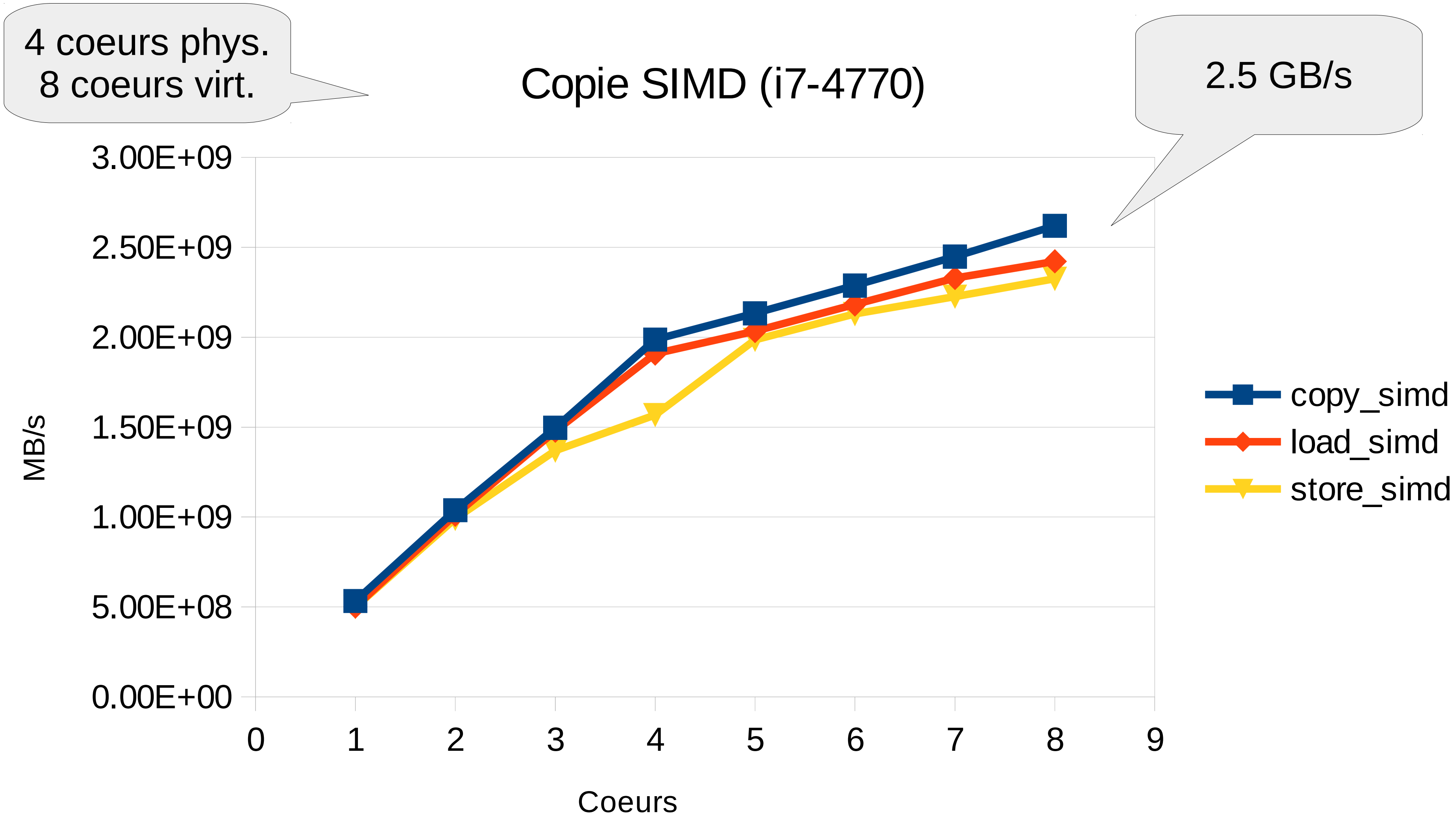
```
QVector<float> &v = m_floats[0];
tbb::parallel_for(tbb::blocked_range<int>(0, m_size),
    [&](tbb::blocked_range<int> &range) {
        for (int i = range.begin(); i < range.end(); i++) {
            volatile float x = v[i];
            (void) x;
        }
    }
);
```

Stores and loads only



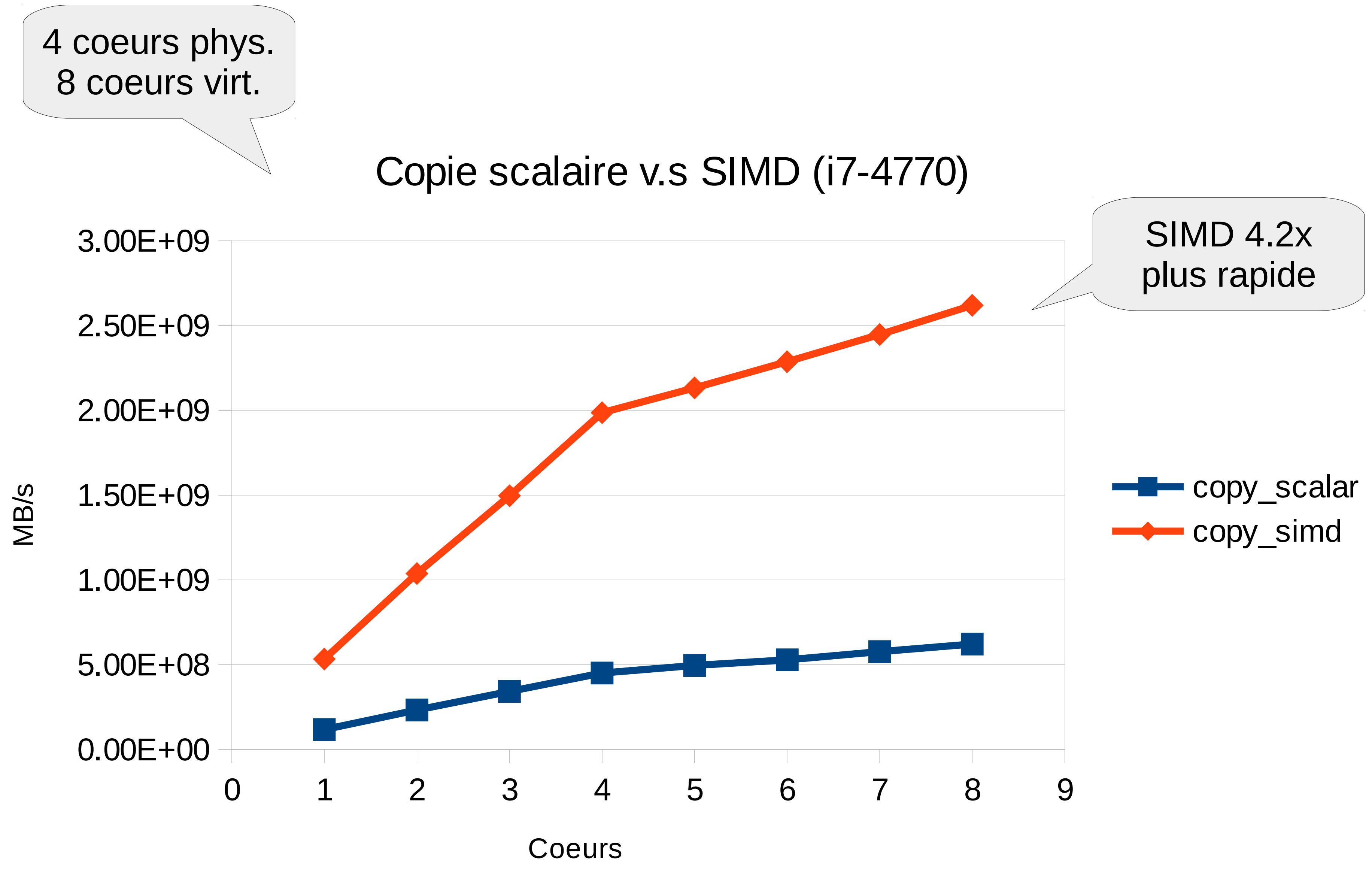
```
For each element of a large vector
load_scalar:  volatile float x = v[i];  -> sizeof(float) * 1
store_scalar: v[i] = x;                 -> sizeof(float) * 1
store_scalar: w[i] = v[i];              -> sizeof(float) * 2
```

POLYTECHNIQUE MONTRÉAL

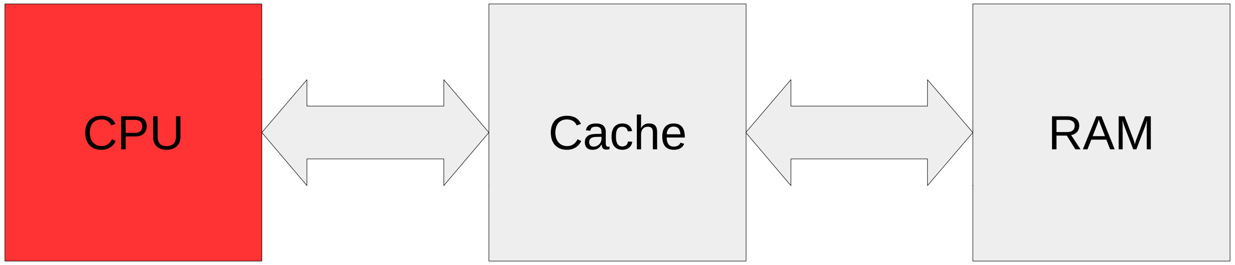


```
__m128 ps = _mm_loadu_ps(src.data() + i);
_mm_storeu_ps(dst.data() + i, ps);
```

```
                        -> sizeof(float) * 4 * 2
```

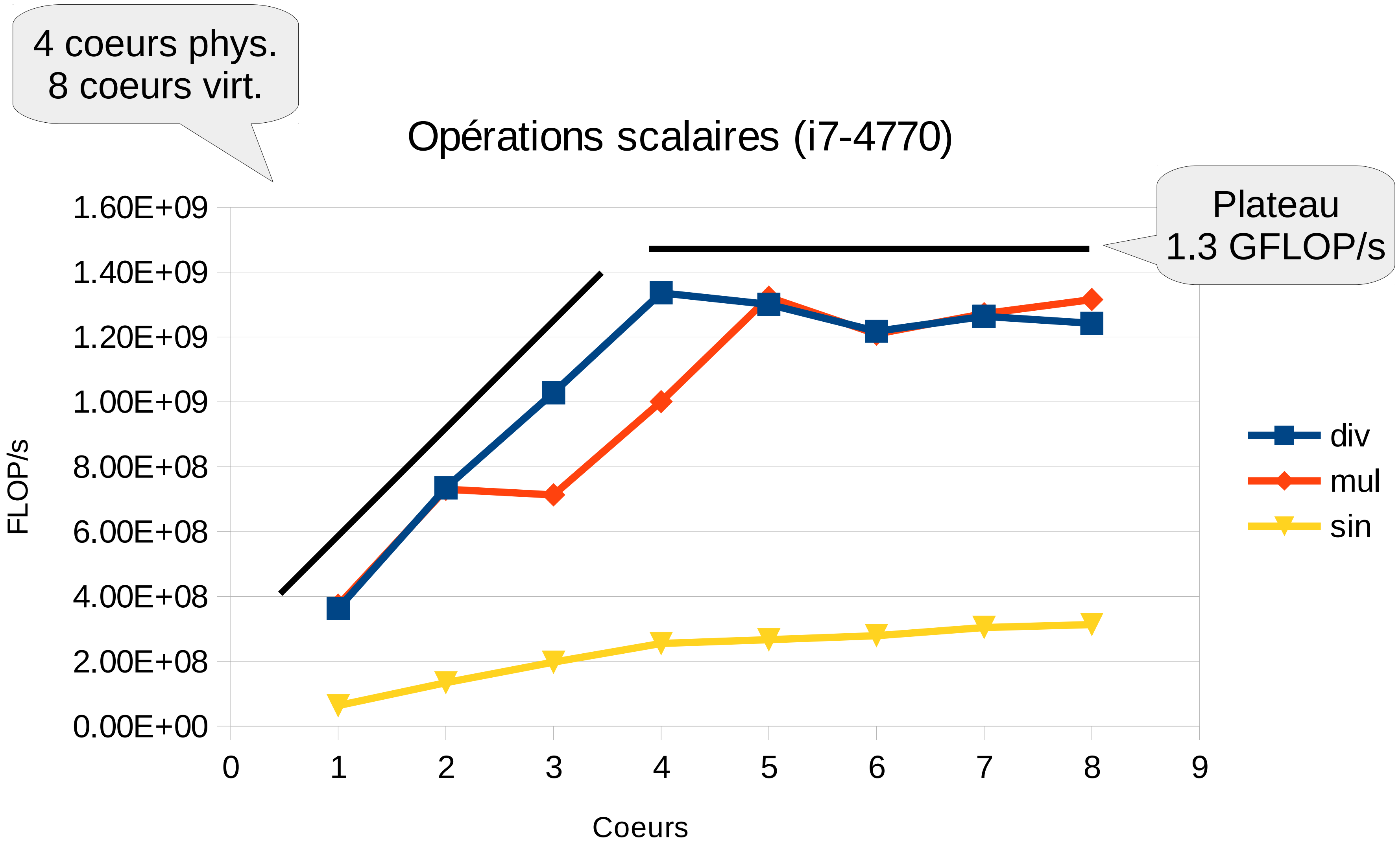

4 coeurs phys.
8 coeurs virt.
Copie scalaire v.s SIMD (i7-4770)
SIMD 4.2x plus rapide
MB/s
3.00E+09
2.50E+09
2.00E+09
1.50E+09
1.00E+09
5.00E+08
0.00E+00
0 1 2 3 4 5 6 7 8 9
Coeurs
copy_scalar
copy_simd

Study of **floating point rate** (FLOP/s)
Intel 8 coeurs HyperThread (4 phys, 8 virt)



```
float cst = 3.1416;
tbb::parallel_for(tbb::blocked_range<int>(0, m_size),
    [&](tbb::blocked_range<int> &range) {
        for (int i = range.begin(); i < range.end(); i++) {
            volatile float val = i * cst;
            (void) val;
        }
    }
);
```

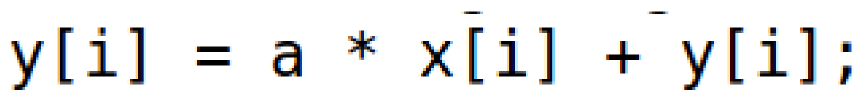
Calcul seulement



```
mul: volatile float x = a * b;          -> 1 FLOP
div: volatile float x = a / b;          -> 1 FLOP
sin: volatile float y = std::sin(x);    -> 5 FLOP
```

Arithmetic intensity

- Ratio: FLOP / octet
- SAXPY:



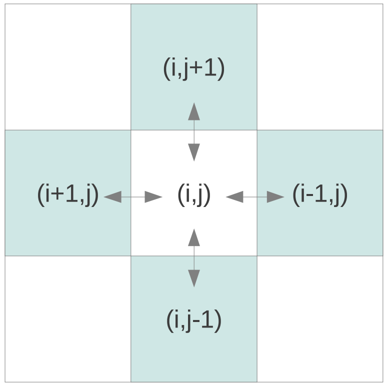
```
y[i] = a * x[i] + y[i];
```

```
- FLOPS: 2 (1 add + 1 mul)
- READ: 2 sizeof(float)
- WRITE: 1 sizeof(float)
- Total: 3 * sizeof(float) = 12
- IA = 2 / 12 = 1/6
```

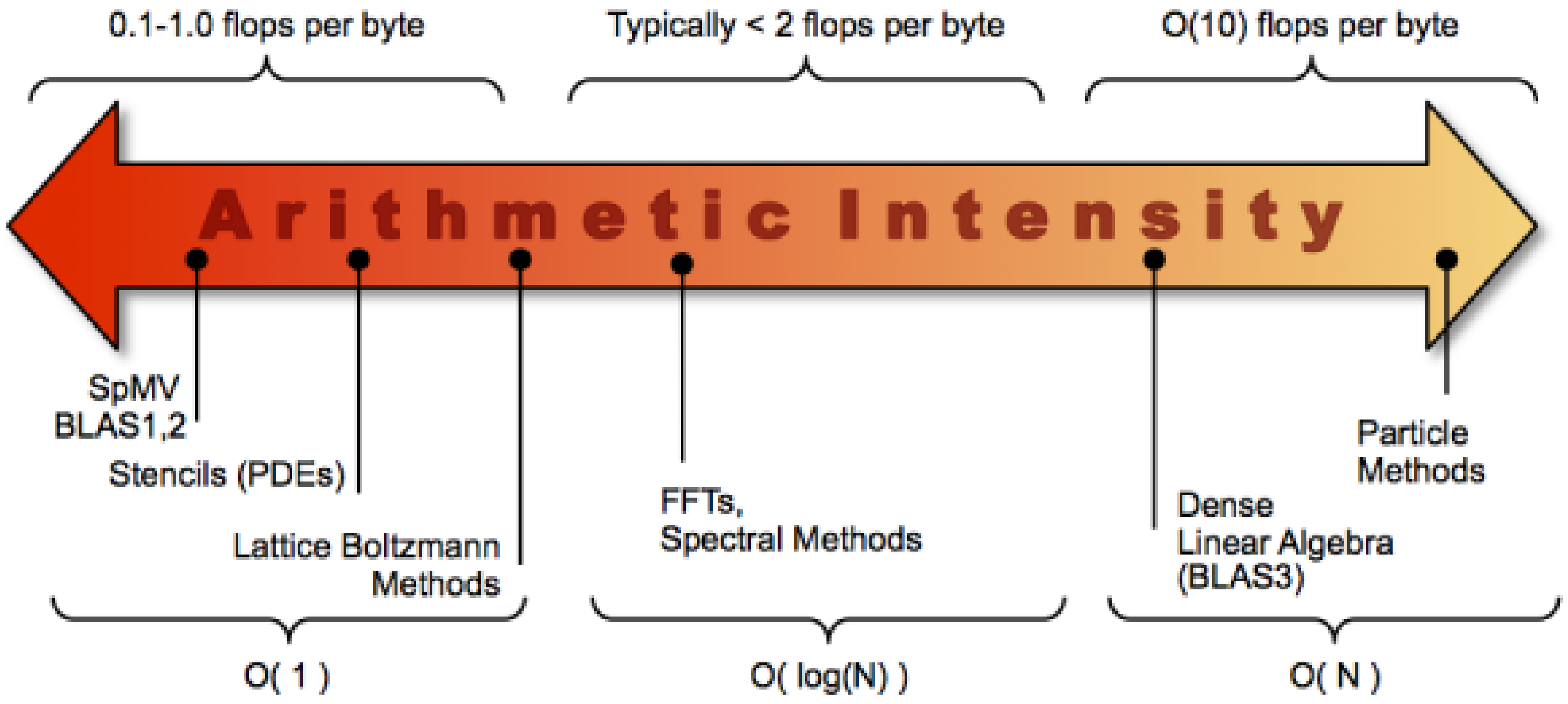
Arithmetic intensity

- Runge-Kutta

```
for (j = 1; j < h - 1; j++) {
    for (i = 1; i < w - 1; i++) {
        g2.dbl[IX2(i,j,w)] =
                g1.dbl[IX2(i,j,w)] + 0.25 * (
                g1.dbl[IX2(i-1,j,w)] +
                g1.dbl[IX2(i+1,j,w)] +
                g1.dbl[IX2(i,j-1,w)] +
                g1.dbl[IX2(i,j+1,w)] - 4 * g1.dbl[IX2(i,j,w)]
                );
    }
}
```

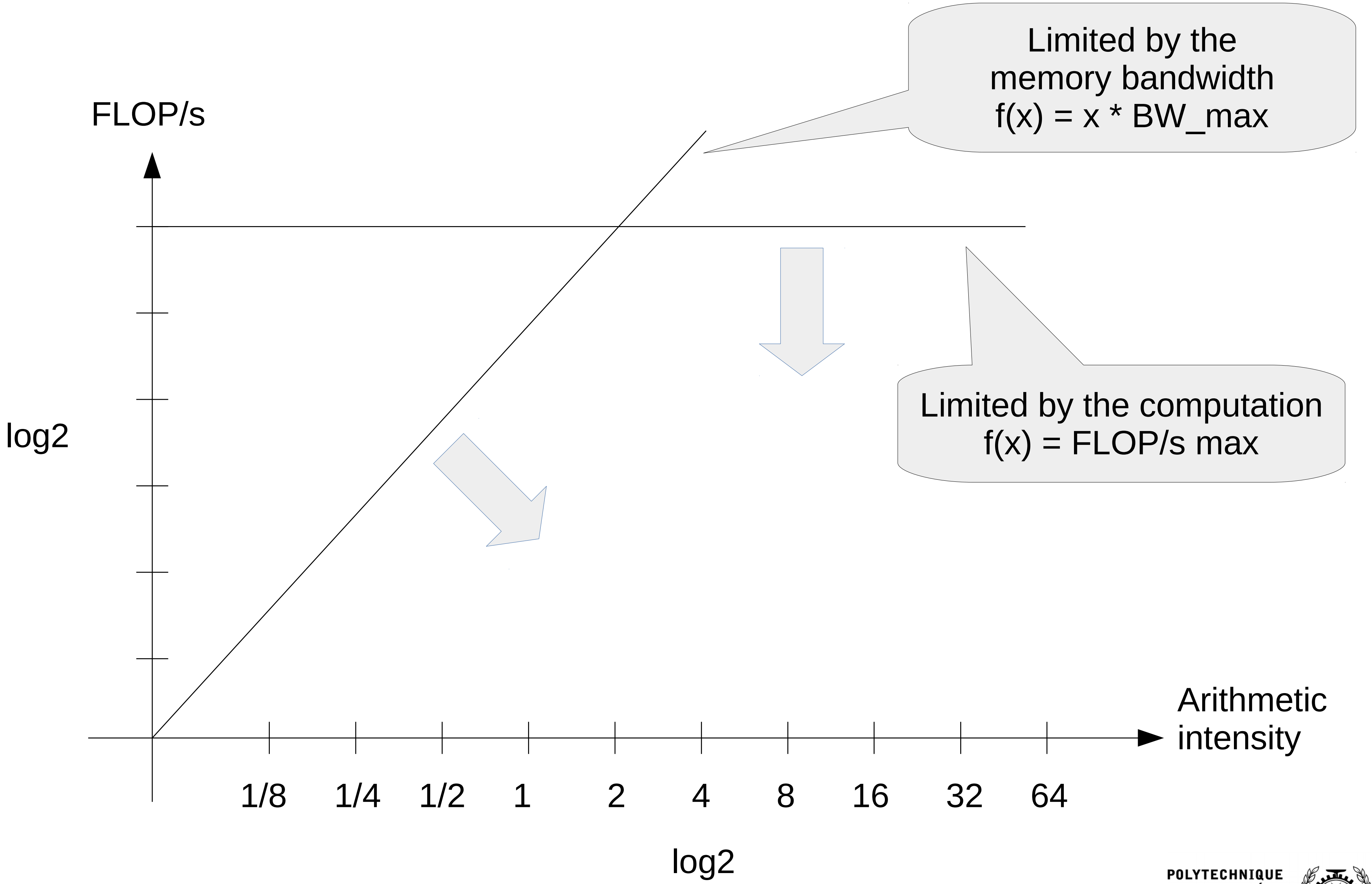


```
- FLOPS: 7 (5 add + 2 mul)
- READ: 5 sizeof(float)
- WRITE: 1 sizeof(float)
- Total: 6 * sizeof(float) = 24
- IA = 7 / 24 ~= 1/3
```

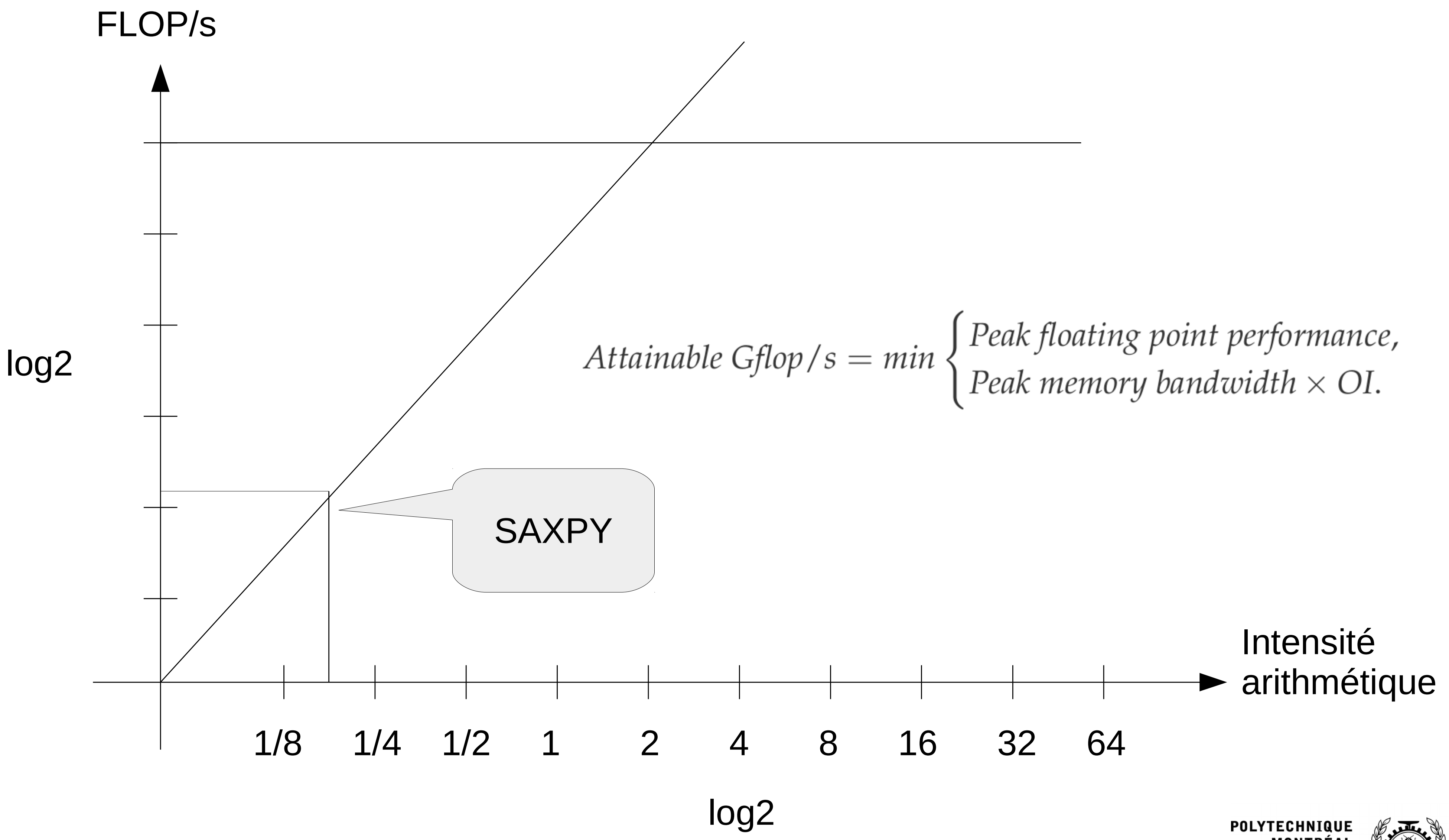


Source: http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

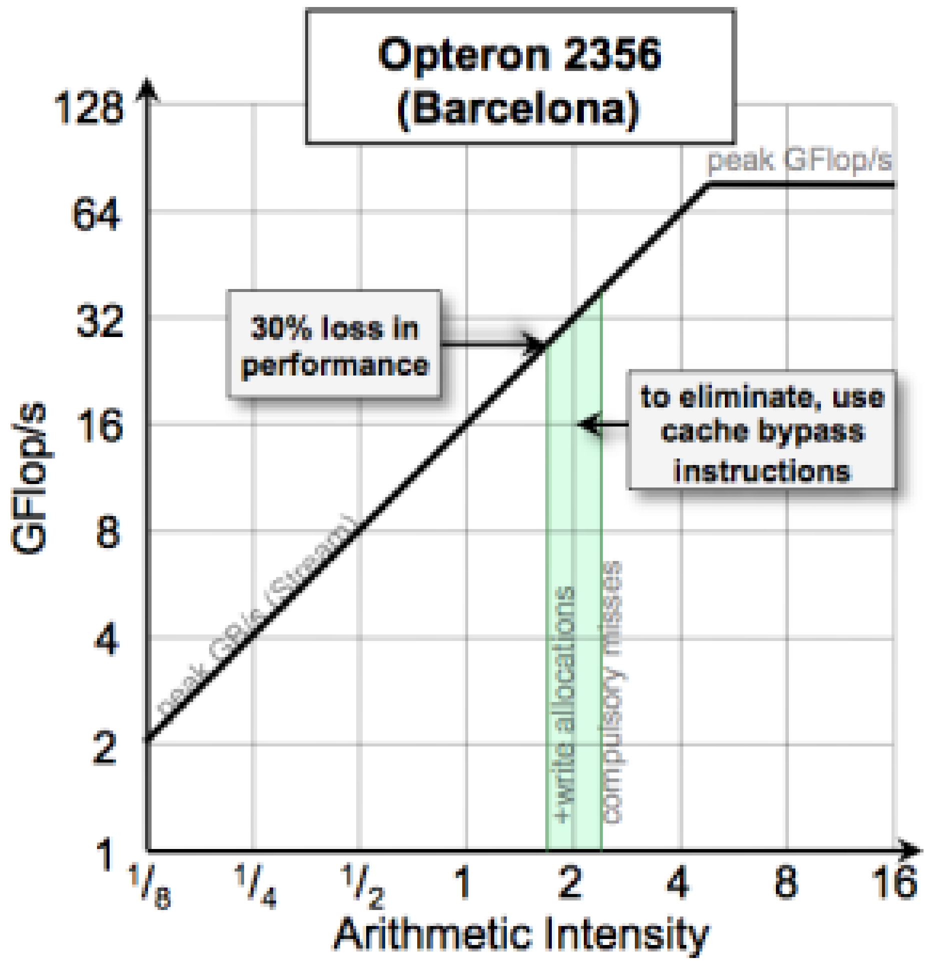
Roofline model

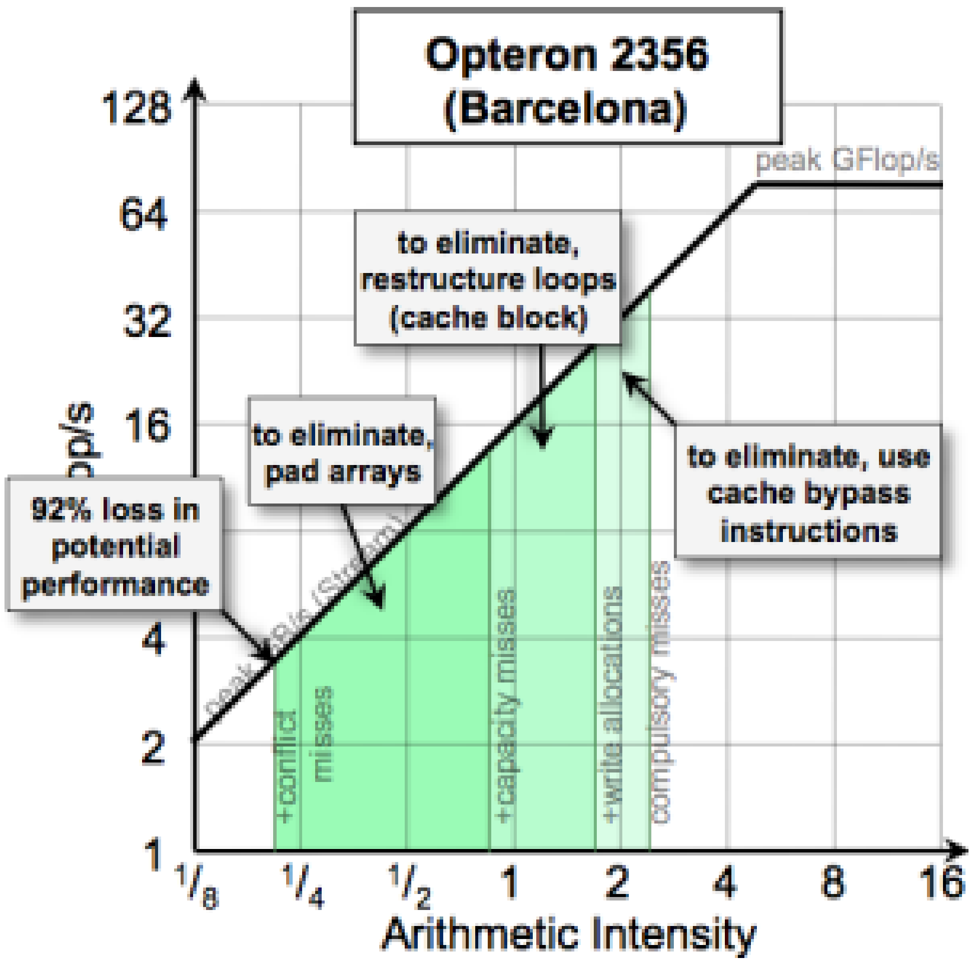


Roofline model



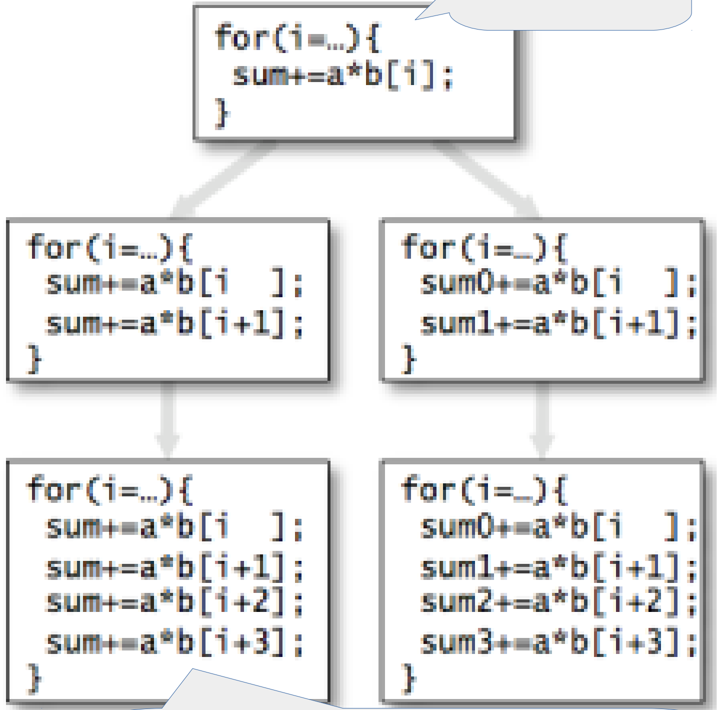
$$\textit{Attainable Gflop/s} = \min \begin{cases} \textit{Peak floating point performance,} \\ \textit{Peak memory bandwidth} \times \textit{OI.} \end{cases}$$





Source: http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

scalaire



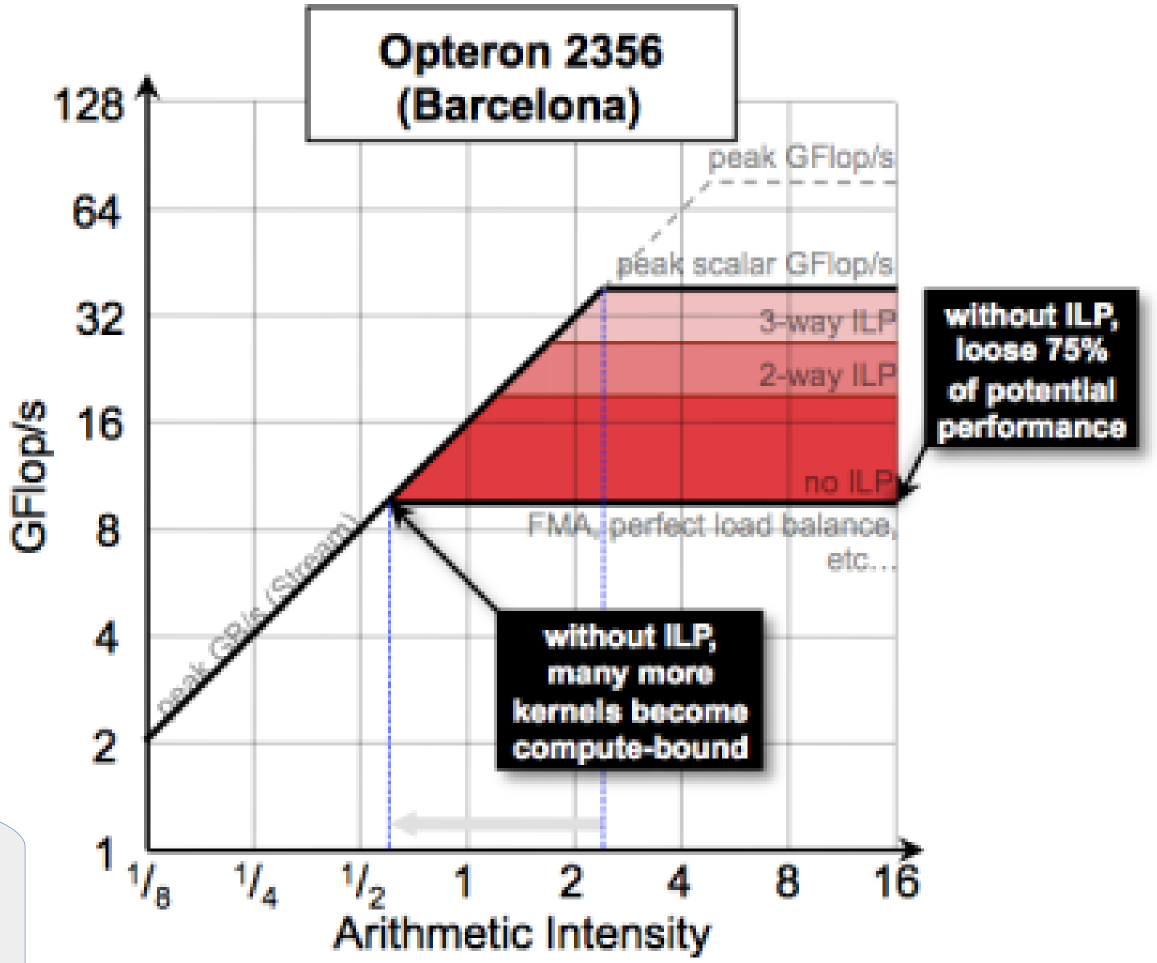
```
for(i=…){
 sum+=a*b[i];
}
```

```
for(i=…){
 sum+=a*b[i  ];
 sum+=a*b[i+1];
}
```

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

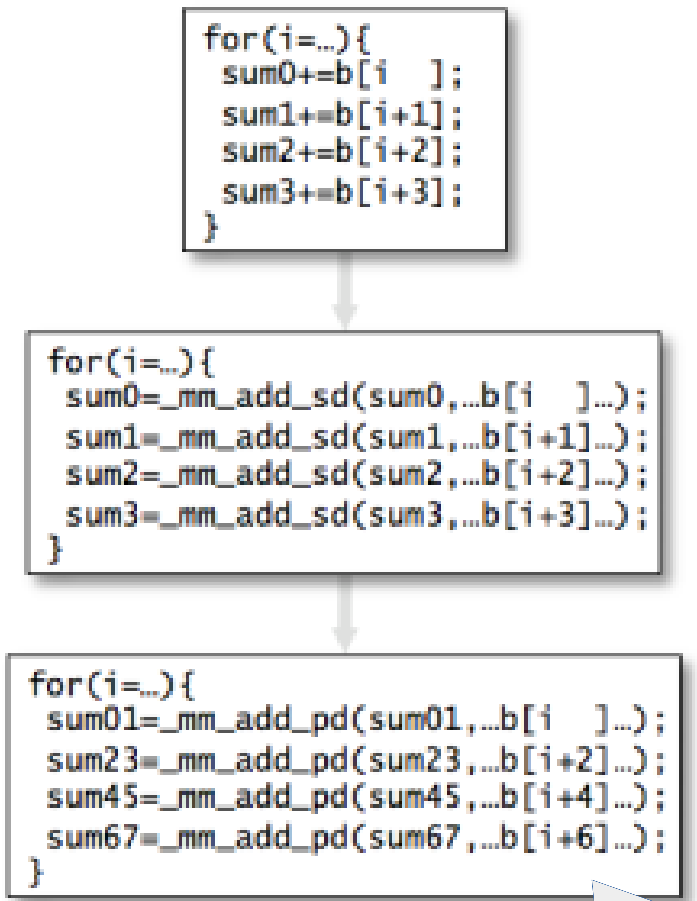
```
for(i=…){
 sum+=a*b[i  ];
 sum+=a*b[i+1];
 sum+=a*b[i+2];
 sum+=a*b[i+3];
}
```

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



Boucle déroulée
Augmente IPL
Seulement s'il n'y a pas de dépendances de données

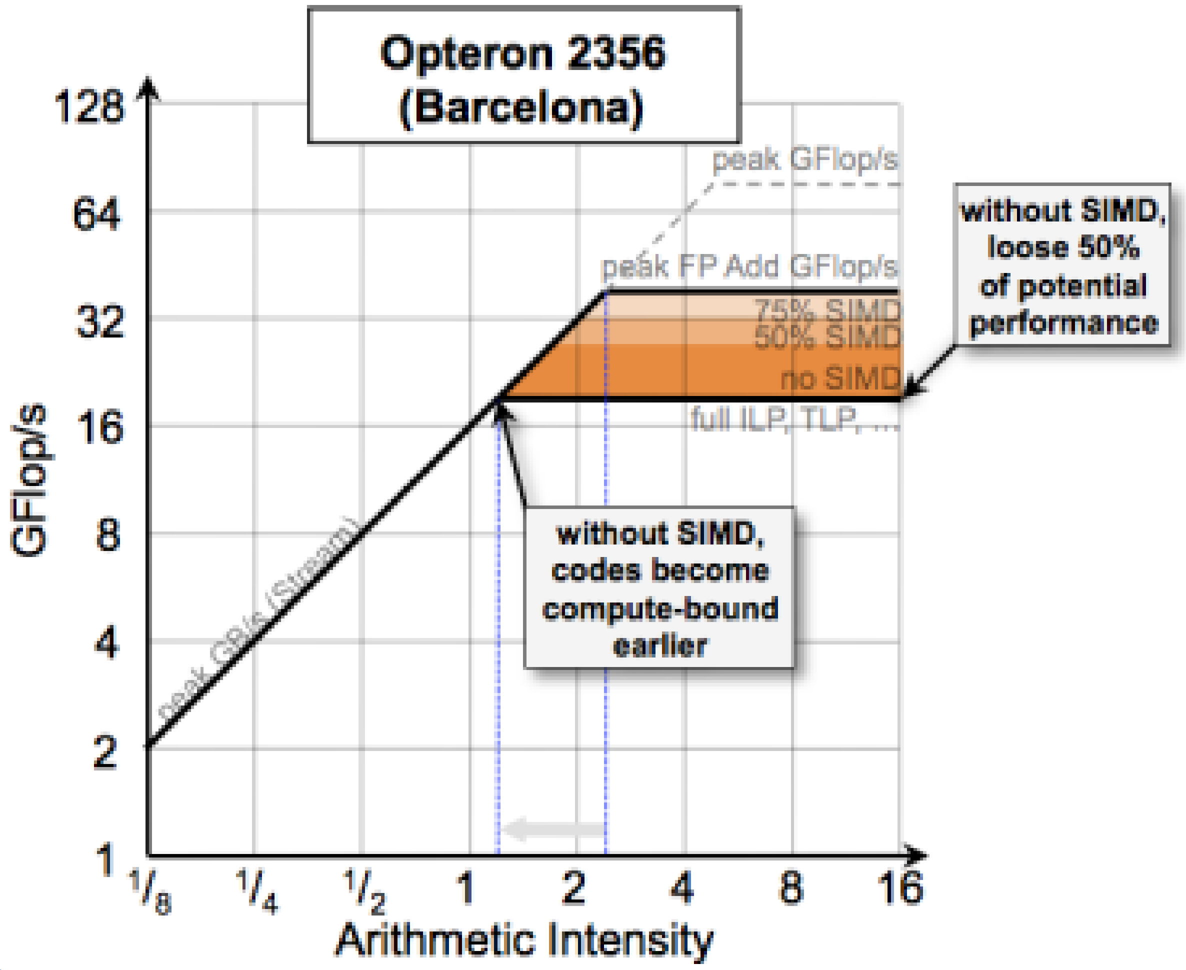
Source: http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/



```
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]…);
}
```



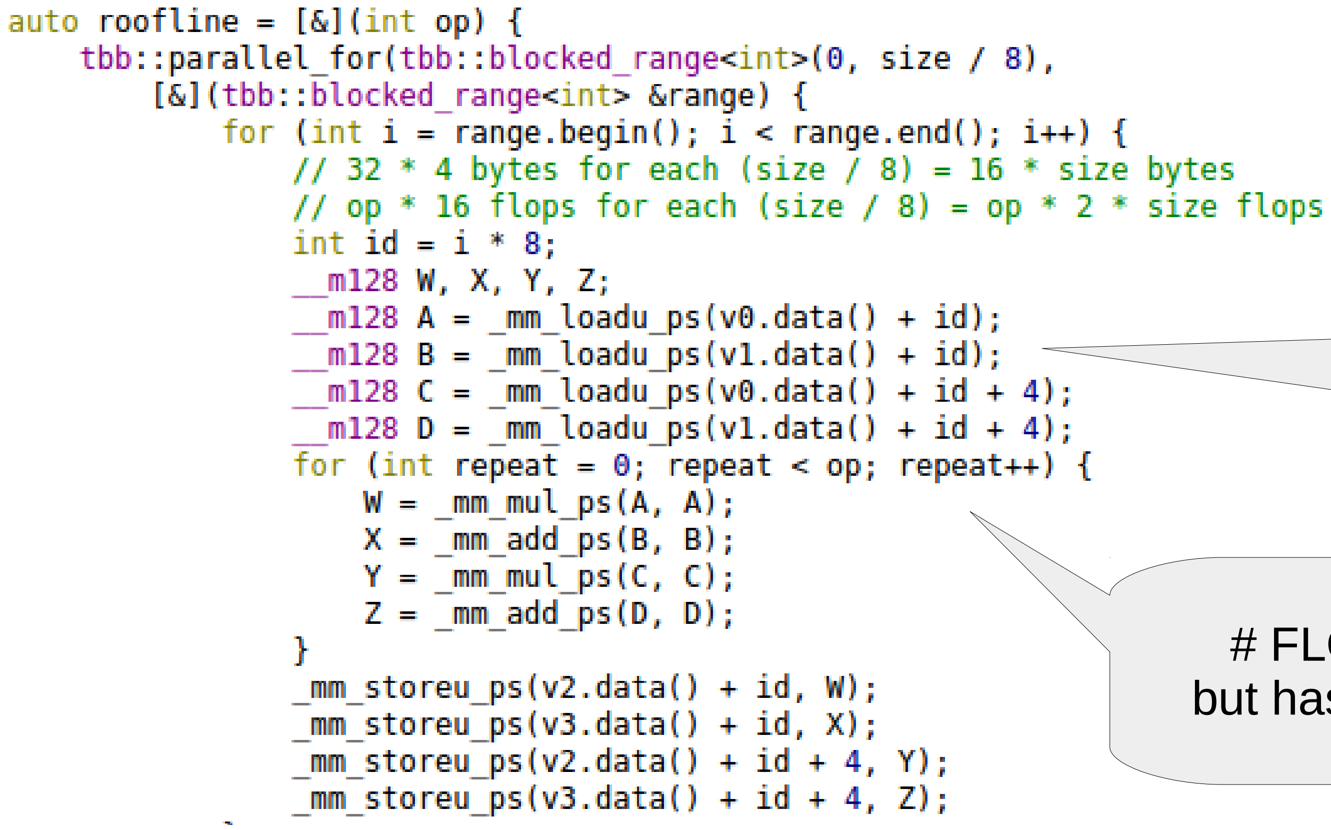
Parallélisme de données
Maintient les ALUs et le bus occupé

Roofline model

- Applies if the working set size is greater than the cache
- Arithmetic intensity is an algorithm property
- Bandwidth and FLOPS are hardware properties

Measure the roofline

- Saturate the memory bus (Copie SIMD)
- Saturate the ALUs (Mul / Add SIMD)
- 51-roofline: theory + actual

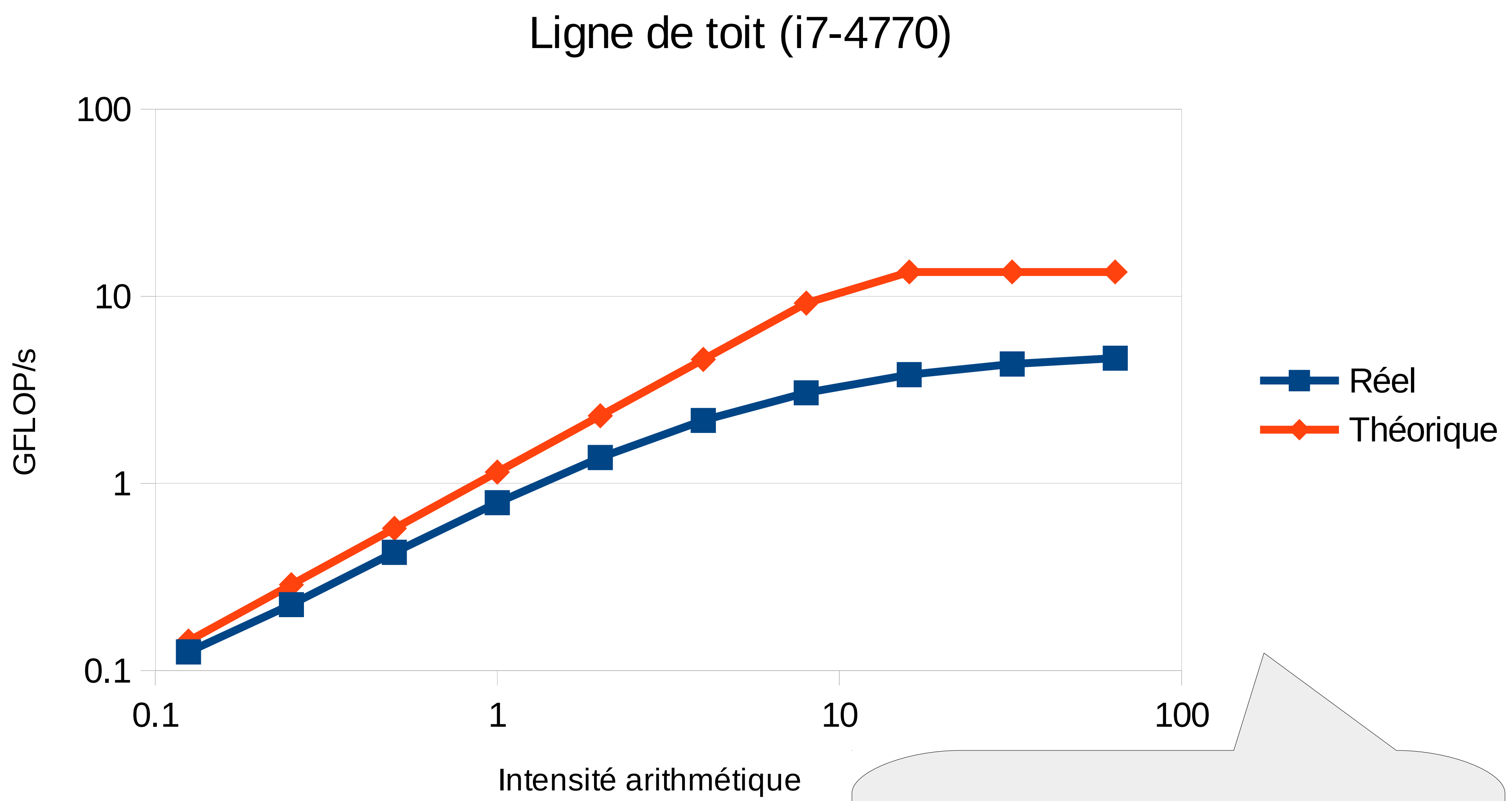


```cpp
auto roofline = [&](int op) {
    tbb::parallel_for(tbb::blocked_range<int>(0, size / 8),
        [&](tbb::blocked_range<int> &range) {
            for (int i = range.begin(); i < range.end(); i++) {
                // 32 * 4 bytes for each (size / 8) = 16 * size bytes
                // op * 16 flops for each (size / 8) = op * 2 * size flops
                int id = i * 8;
                __m128 W, X, Y, Z;
                __m128 A = _mm_loadu_ps(v0.data() + id);
                __m128 B = _mm_loadu_ps(v1.data() + id);
                __m128 C = _mm_loadu_ps(v0.data() + id + 4);
                __m128 D = _mm_loadu_ps(v1.data() + id + 4);
                for (int repeat = 0; repeat < op; repeat++) {
                    W = _mm_mul_ps(A, A);
                    X = _mm_add_ps(B, B);
                    Y = _mm_mul_ps(C, C);
                    Z = _mm_add_ps(D, D);
                }
                _mm_storeu_ps(v2.data() + id, W);
                _mm_storeu_ps(v3.data() + id, X);
                _mm_storeu_ps(v2.data() + id + 4, Y);
                _mm_storeu_ps(v3.data() + id + 4, Z);
```

IPL +
SIMD +
ADD/MUL balanced

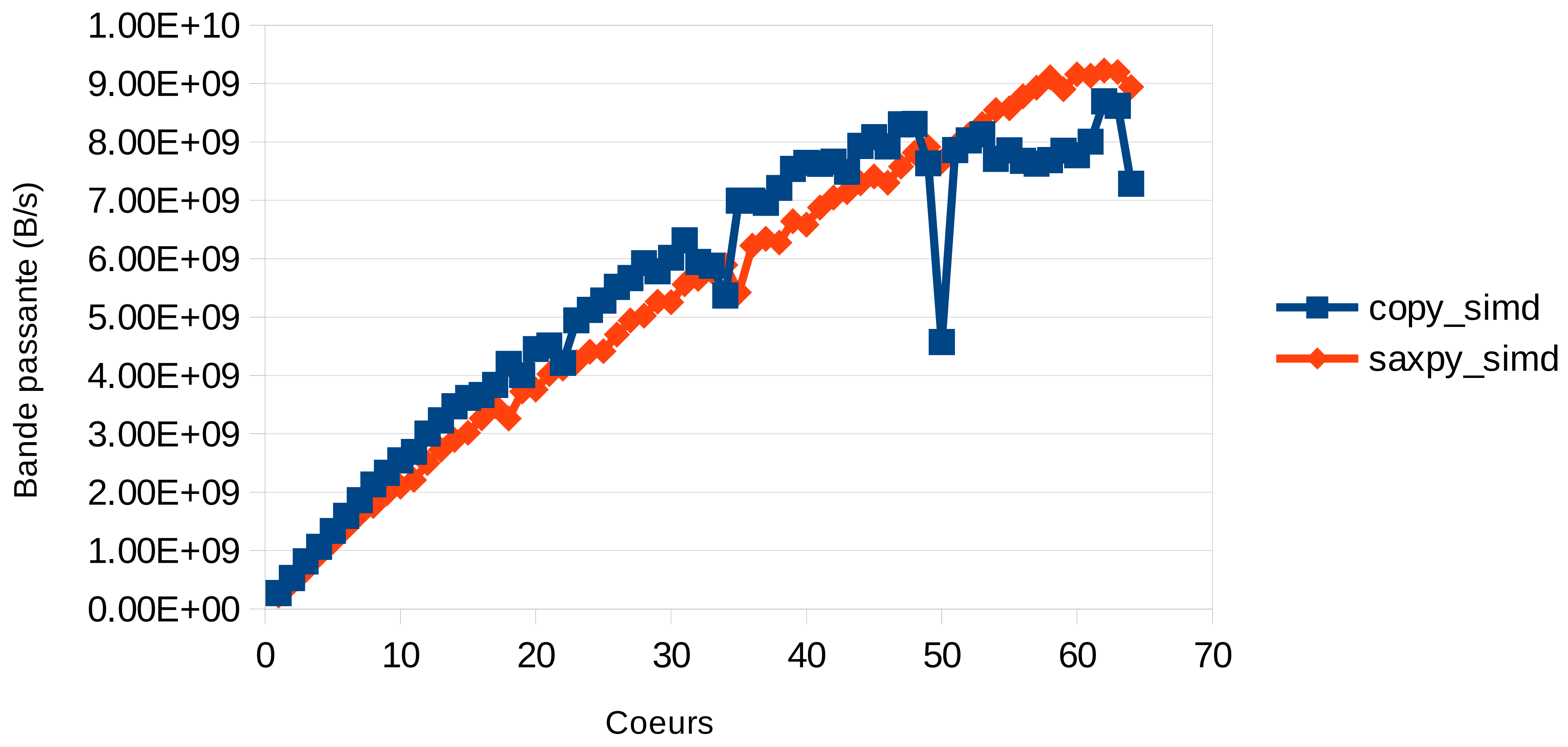
# FLOPS is variable,
but has branch hazard...

POLYTECHNIQUE
MONTRÉAL

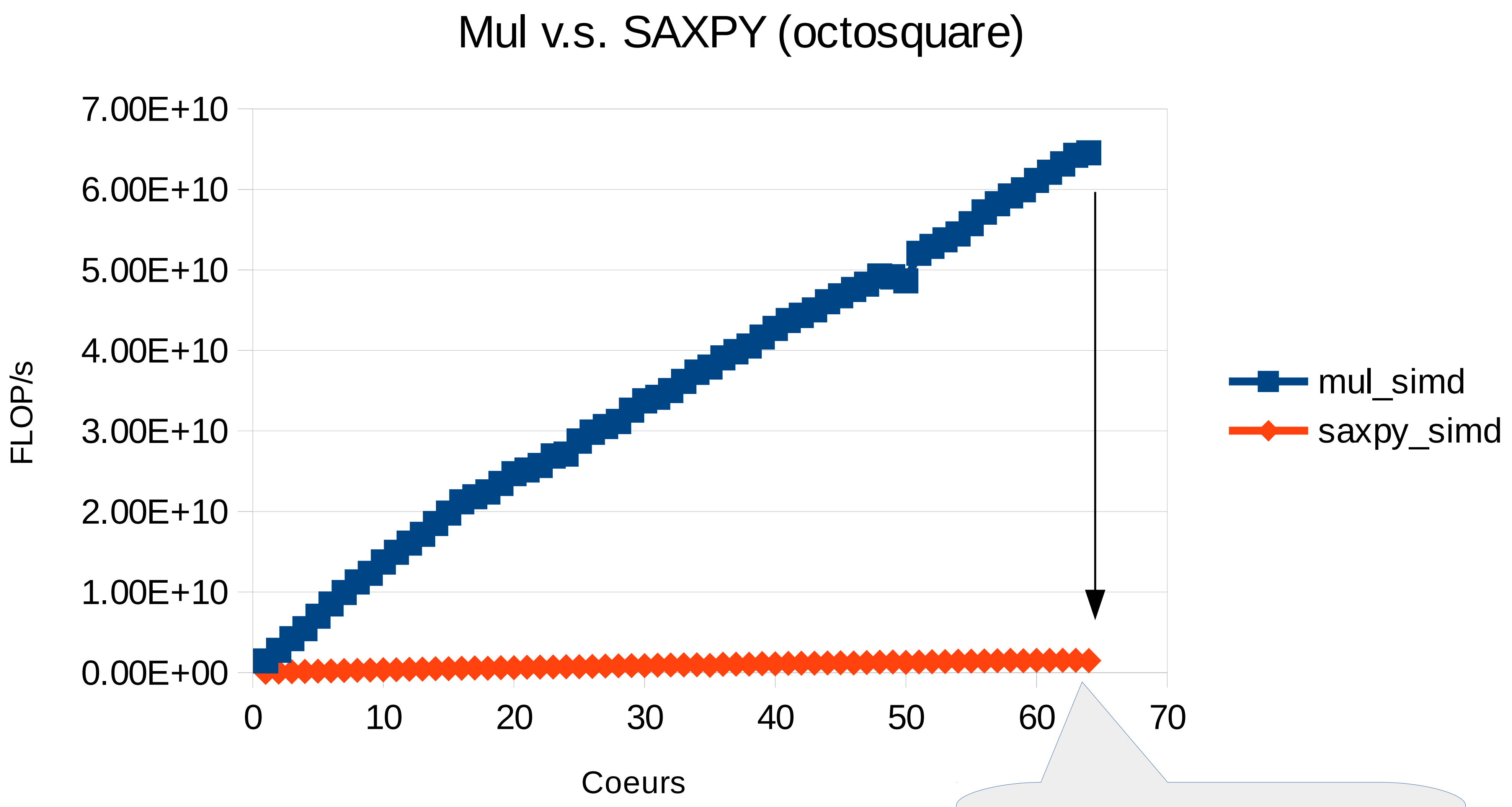


Max theory: 13.5 GFLOP/s
Max real: 4.7 GFLOP/s
Real is 2.8x slower than expected!

Copie v.s. SAXPY (octosquare)



POLYTECHNIQUE MONTRÉAL



43x fois moins de FLOP/s que le max théorique

Observations

- Computation is *cheap*
- Communication is costly
- Optimizing computation is useless if the memory bandwidth is the bottleneck

Reduce memory bandwidth usage

- SIMD: essential
  - Instructions must be loaded from the cache
  - Less instructions means more bandwidth can be used for data
- Loop fusion: use the data while it is in registers
- Divide data in blocks that fits in the cache
- Recompute data instead of fetching in memory

Loop fusion

```
void low_ai()
{
    // AI: 1/6=2 flops/12 bytes
    for(int i=0; i < N; i++){
        y[i]=a*x[i] + y[i];
    }

    // AI: 1/4=1 flop/4 bytes
    double sum=0;
    for(int i=0; i < N; i++) {
        sum+=y[i];
    }
}
```

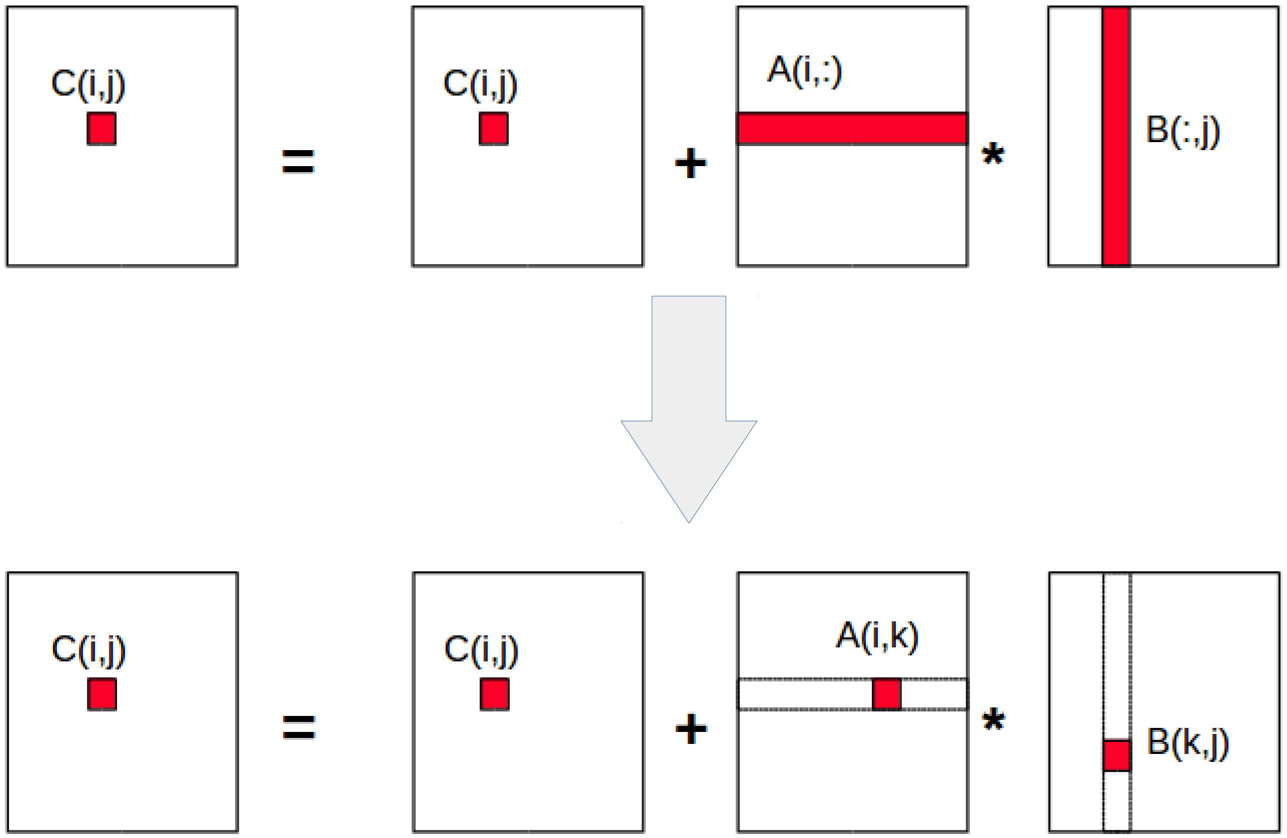
fusion

```
void better_ai()
{
    // AI: 1/4
    // 12 bytes
    // 3 flops
    double sum=0;
    for(int i=0; i < N; i++){
        y[i]=a*x[i] + y[i];
        sum +=y[i];
    }
}
```

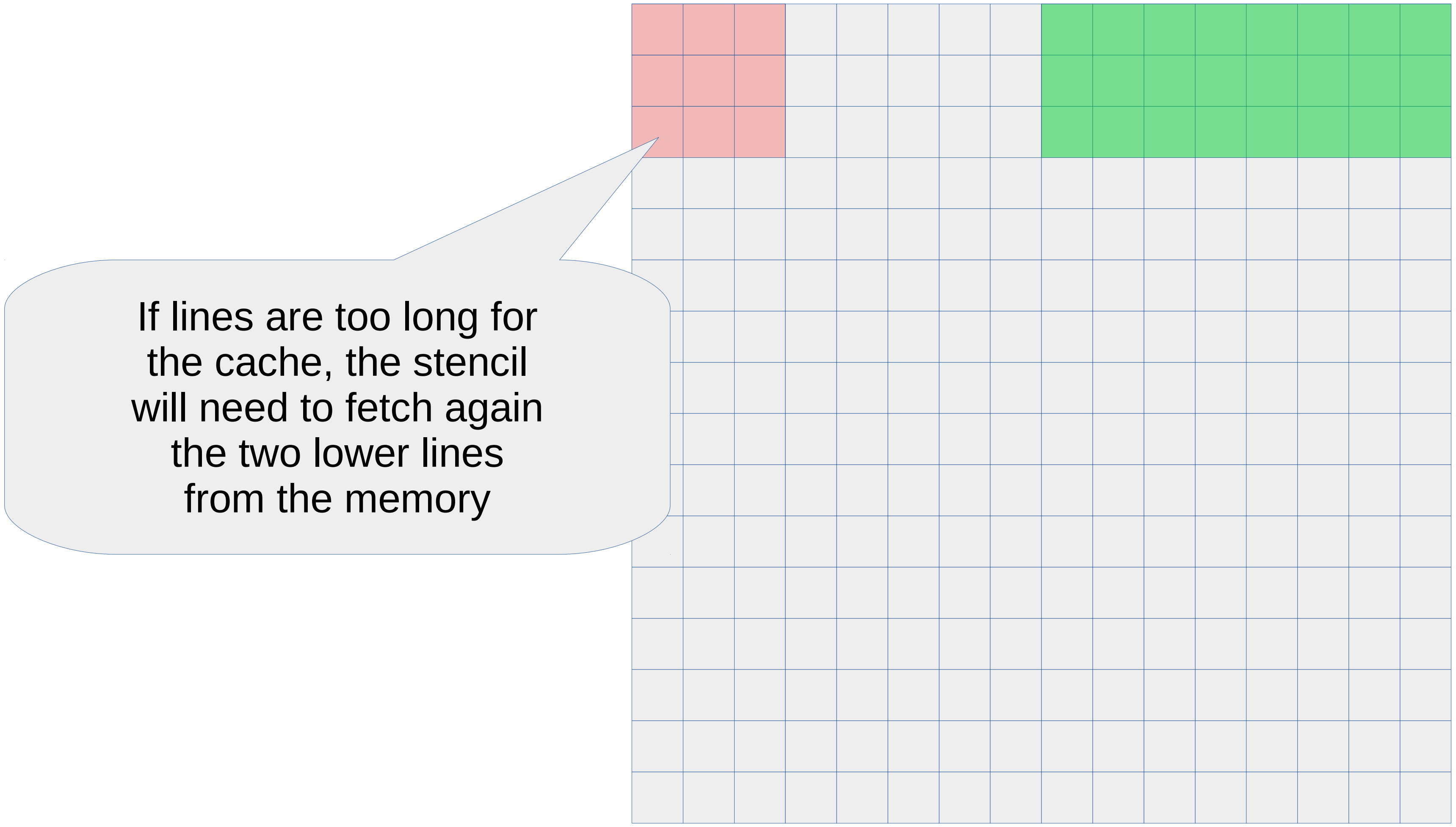
If the vector doesn’t fit in the cache, the next loop will fetch it again.

Reuse y[i] while it is in a register

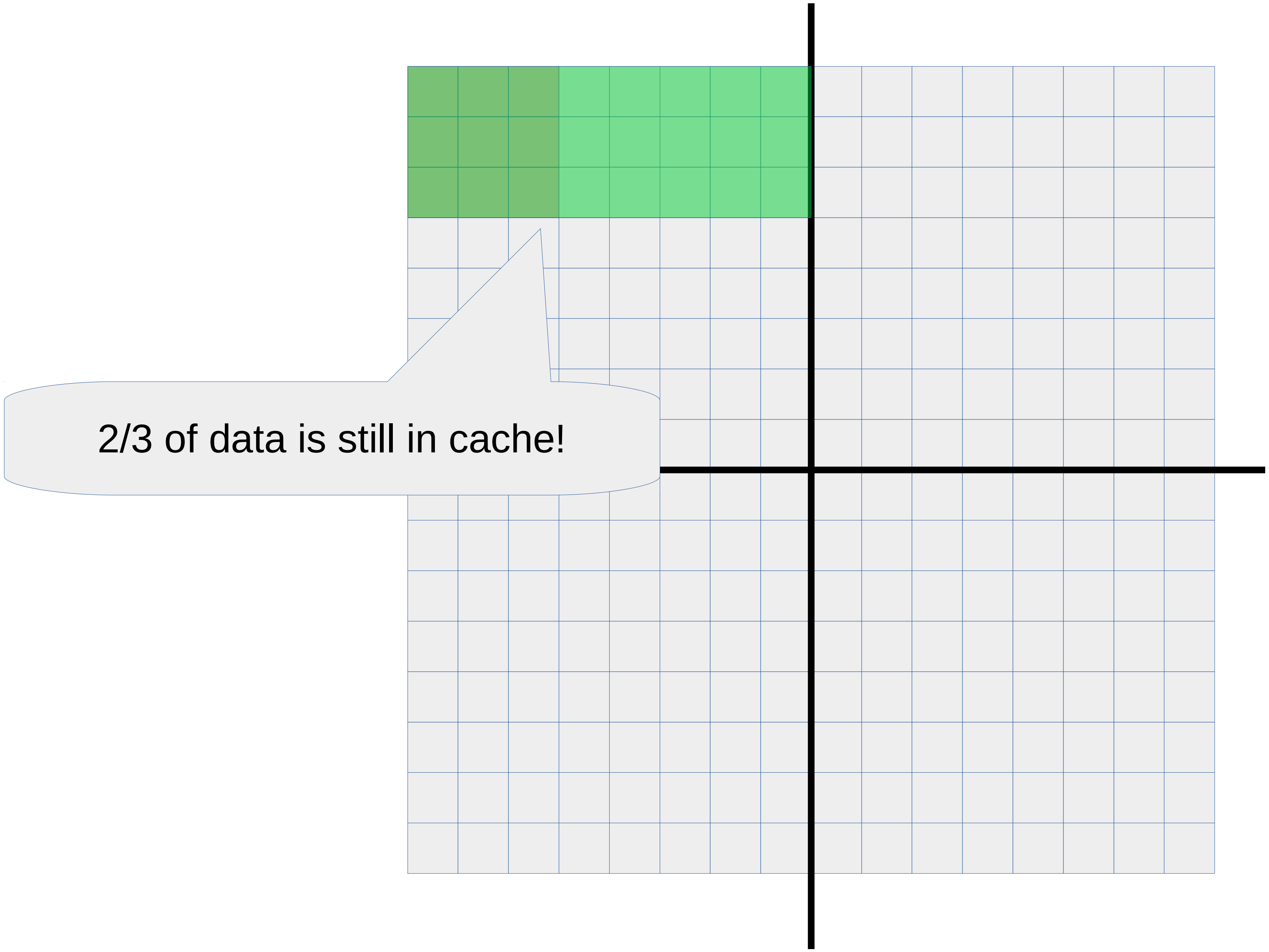
Divide data as block



Divide data as block

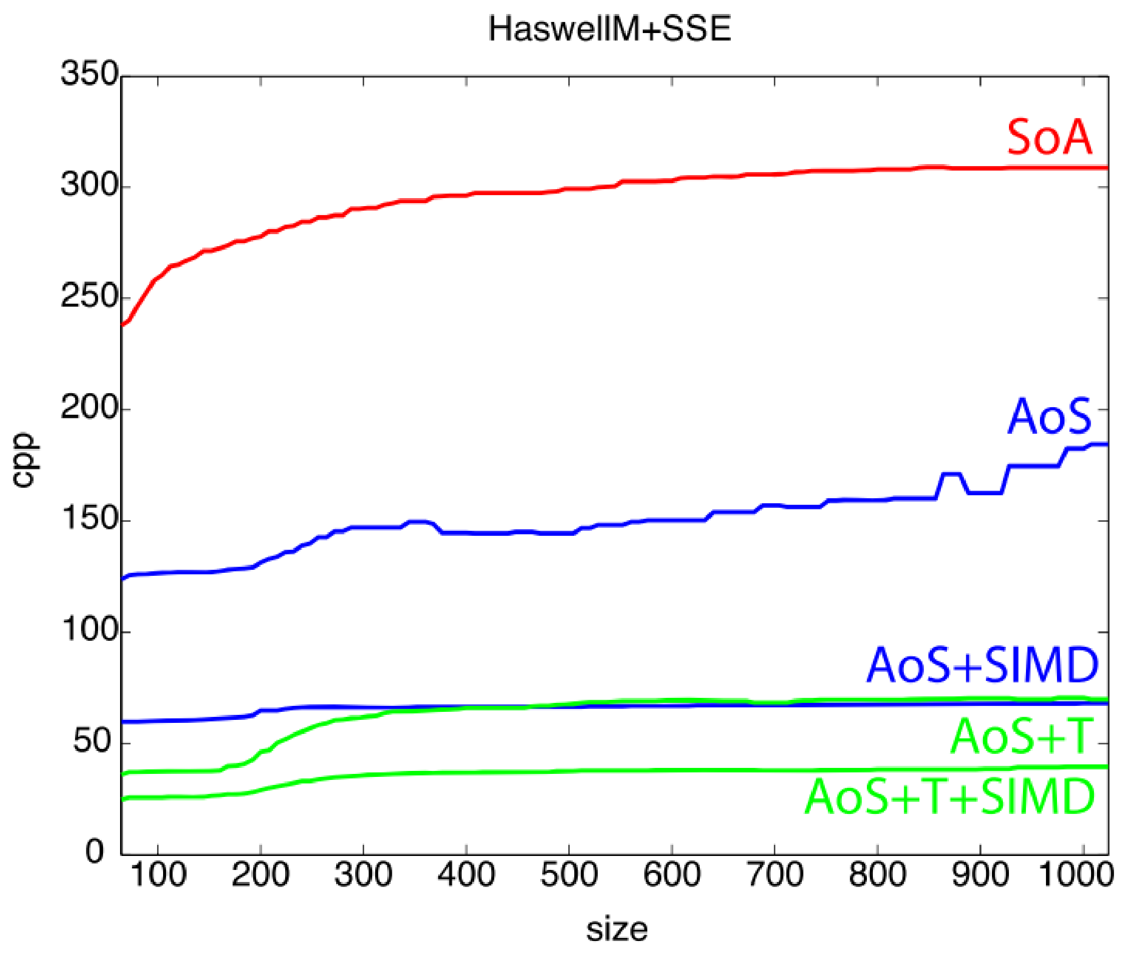

If lines are too long for the cache, the stencil will need to fetch again the two lower lines from the memory

Divide data as block



POLYTECHNIQUE
MONTRÉAL

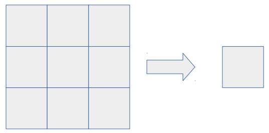
Example: compacting data

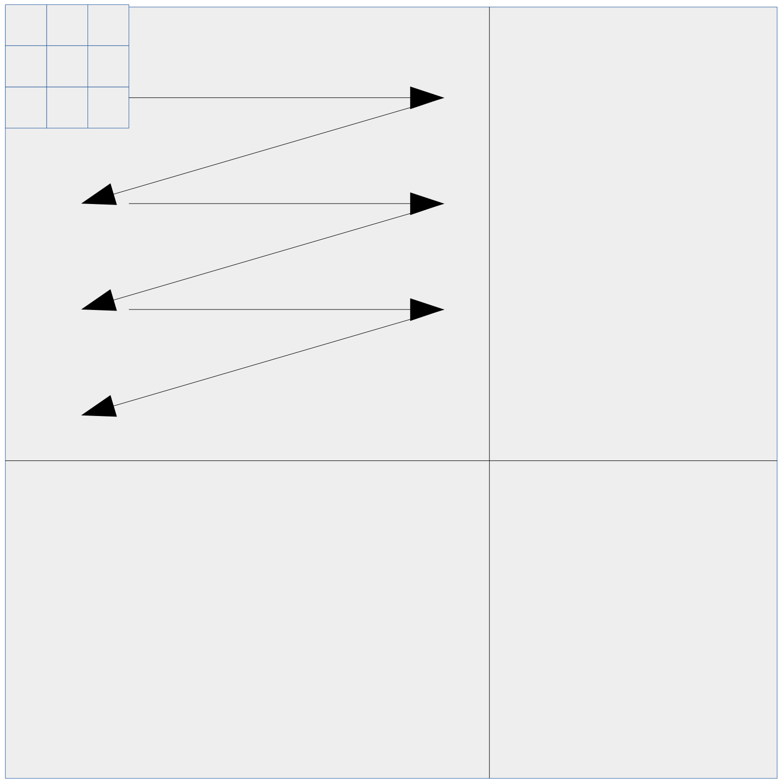


Source: "Covariance tracking: architecture optimizations for embedded systems" A. Roméro, L. Lacassagne, M. Gouiffès, A. Hassan Zahraee, 2014

Division 2D

- blocked_range2d<T>(0, width, bsx, 0, height, bsy)
- Moving average of 2D data (image)





Example using TBB

```
void mean3_tbb2d(vector<float> &m,
                 vector<float> &x,
                 int width, int height)
{
 tbb::parallel_for(
  tbb::blocked_range2d<uint>(1, width - 1, 1, height - 1),
   [&](tbb::blocked_range2d<uint> &r) {
    for (uint i=r.rows().begin(); i<r.rows().end(); i++){
     for (uint j=r.cols().begin(); j<r.cols().end(); j++){
      uint idx0 = (i - 1) * width + j;
      uint idx1 = (i + 0) * width + j;
      uint idx2 = (i + 1) * width + j;
      m[idx1] = (x[idx0 - 1] + x[idx0] + x[idx0 + 1] +
                 x[idx1 - 1] + x[idx1] + x[idx1 + 1] +
                 x[idx2 - 1] + x[idx2] + x[idx2 + 1]) *
                 (1/9.0f);
      }
     }
    }
   );
}
```

POLYTECHNIQUE MONTRÉAL

Activities

- 50-pmu: how to use hardware performance counters
- 51-roofline: find the roofline of your computer
- 52-cache-size: find the cache size of your computer
- Use LTTng to record hardware performance counter as event context