SYCL

Techniques d’optimisation

SYCL

Plusieurs types de kernel

- Simples
- ND-range
  - Notion de groupes et de sous groupes
- Hiérarchiques

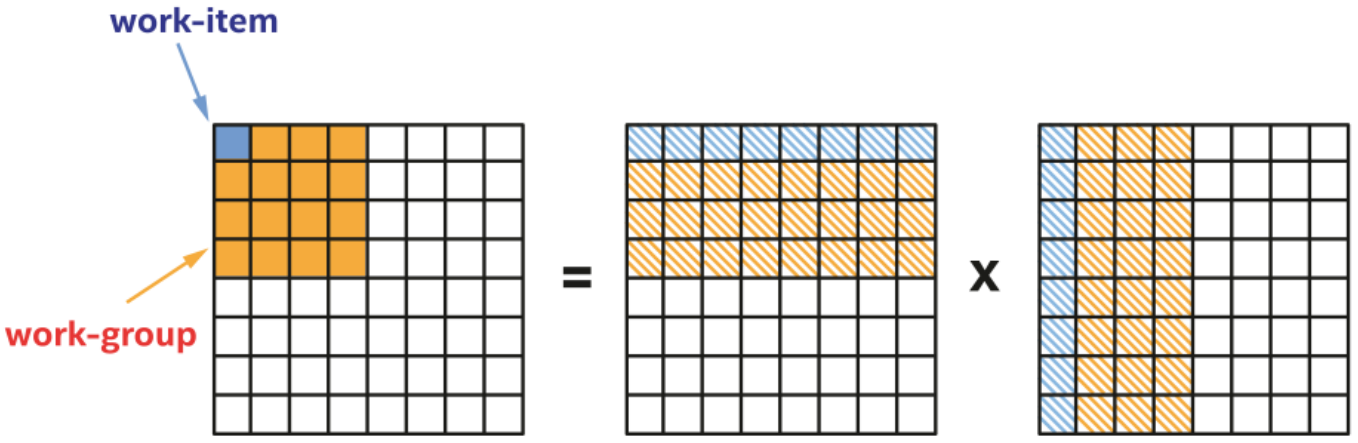
Kernels ND-range

Le nd_range définit l'espace d'indexation d'un groupe de travail ainsi que l'espace d'indexation global. Il est passé à parallel_for pour exécuter un kernel sur un ensemble de work- items.

```
range global{N, N};
range local{B, B};
h.parallel_for(nd_range{global, local}, [=](nd_item<2> it) {
  int j = it.get_global_id(0);
  int i = it.get_global_id(1);

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

Kernels ND-range

Le fait de regrouper work-items de cette manière garantit la localité de l'accès et, souvent, améliore l’utilisation de la mémoire cache.

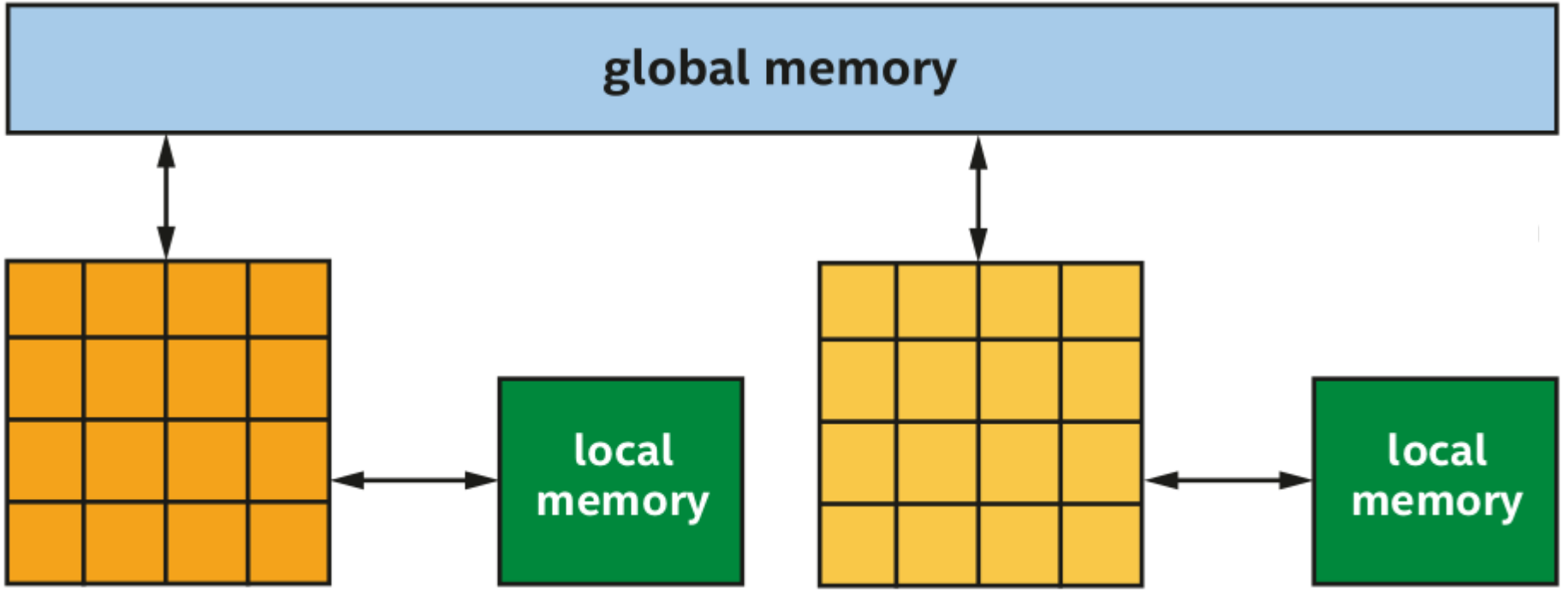


Kernels ND-range

Les work-items dans un work-groupe ont accès à la mémoire locale du groupe qui est directement mise en correspondance avec de la mémoire rapide sur certains devices.

Les work-items dans un groupe peuvent utiliser des instructions de synchronisation rapides.

Kernels ND-range



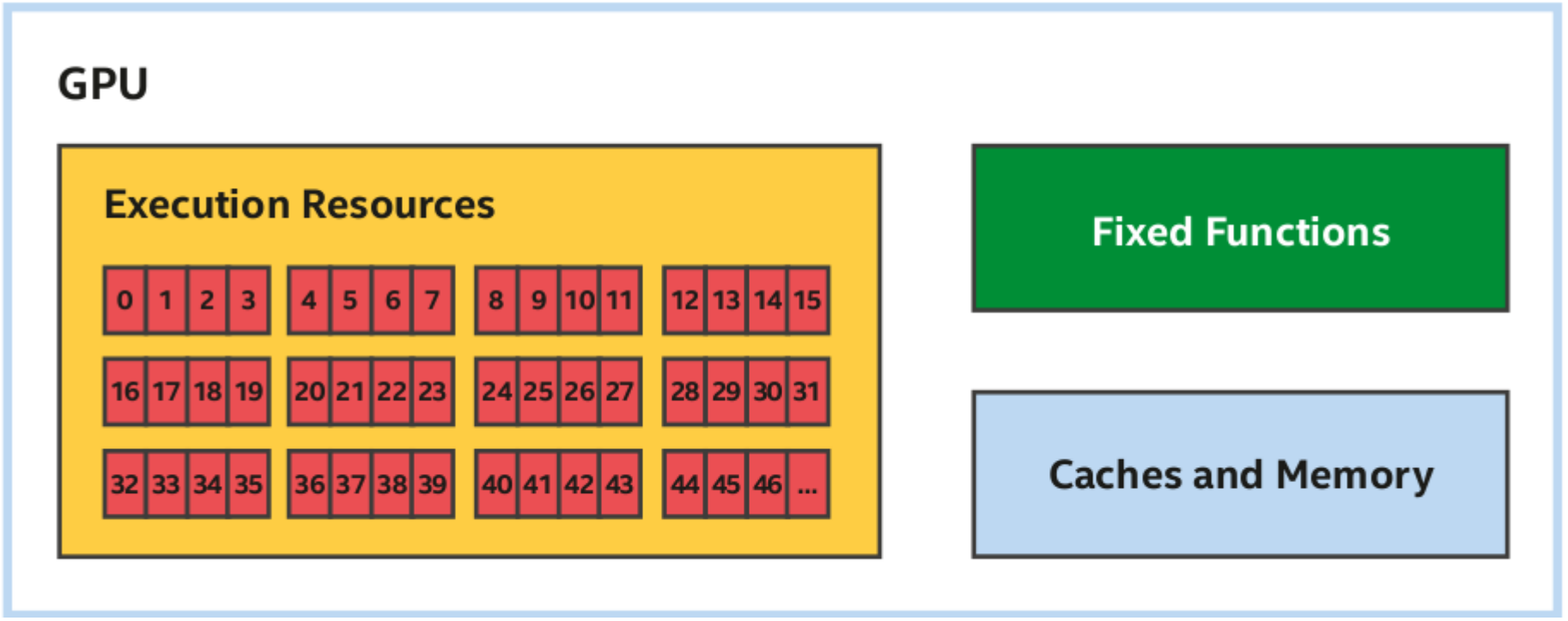
Kernels ND-range

```
local_accessor<int> tileA{tile_size, h};

h.parallel_for(
  nd_range<2>{{M, N}, {1, tile_size}}, [=](nd_item<2> item) {
    // Indices in the global index space:
    int m = item.get_global_id()[0];
    int n = item.get_global_id()[1];

    // Index in the local index space:
    int i = item.get_local_id()[1];

    T sum = 0;
    for (int kk = 0; kk < K; kk += tile_size) {
      // Load the matrix tile from matrix A, and synchronize
      // to ensure all work-items have a consistent view
      // of the matrix tile in local memory.
      tileA[i] = matrixA[m][kk + i];
      item.barrier();

      // Perform computation using the local memory tile, and
      // matrix B in global memory.
      for (int k = 0; k < tile_size; k++)
        sum += tileA[k] * matrixB[kk + k][n];

      // After computation, synchronize again, to ensure all
      // reads from the local memory tile are complete.
      item.barrier();
    }

    // Write the final result to global memory.
    matrixC[m][n] = sum;
});
```

Kernels Hierarchiques

```
range group_size{16};
range num_groups = size / group_size;

h.parallel_for_work_group(num_groups, group_size, [=](group<1> group) {
  // This variable is declared at work-group scope, so
  // it is allocated in local memory and accessible to
  // all work-items.
  int localIntArr[16];

  // There is an implicit barrier between code and variables
  // declared at work-group scope and the code and variables
  // at work-item scope.

  group.parallel_for_work_item([&](h_item<1> item) {
    auto index = item.get_global_id();
    auto local_index = item.get_local_id();

    // The code at work-item scope can read and write the
    // variables declared at work-group scope.
    localIntArr[local_index] = index + 1;
    data_acc[index] = localIntArr[local_index];
  });
});
```

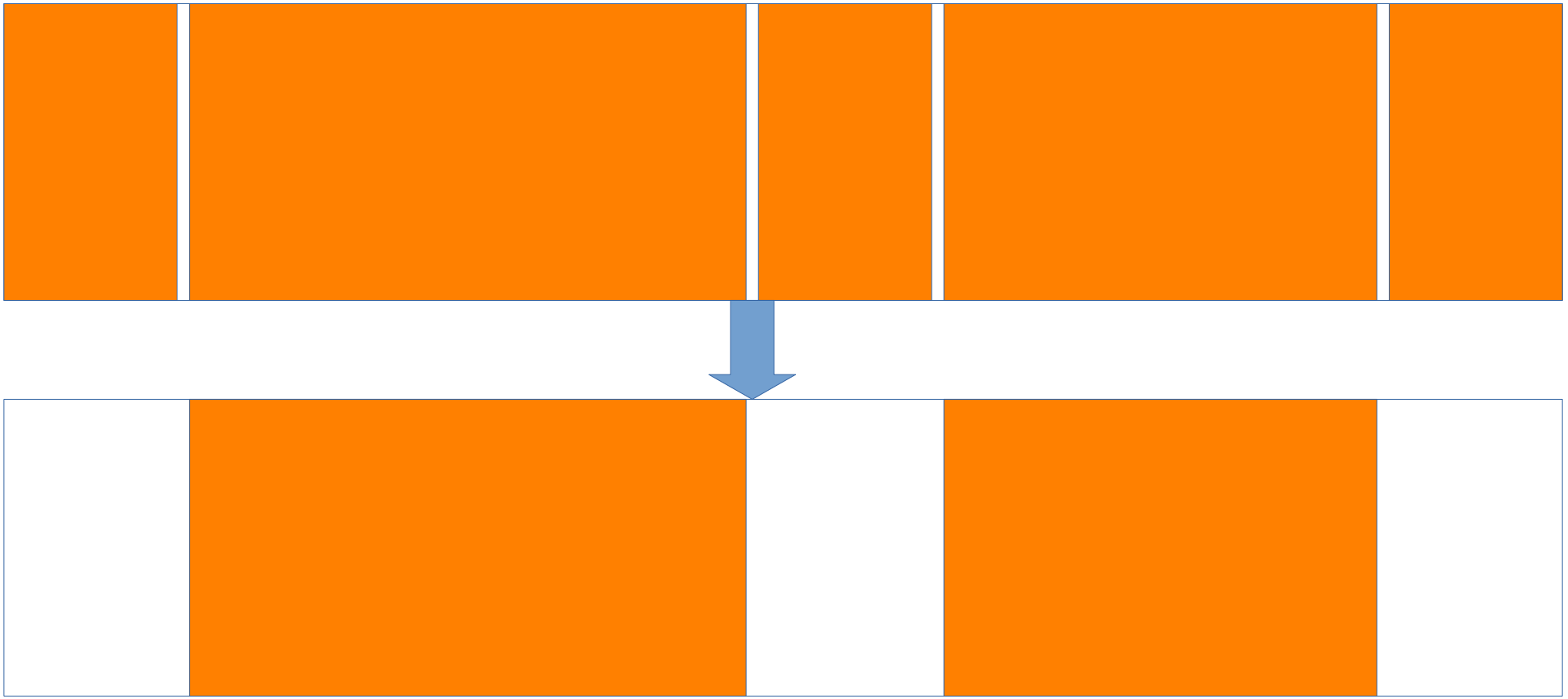
GPU


GPU
Execution Resources
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 ...
Fixed Functions
Caches and Memory

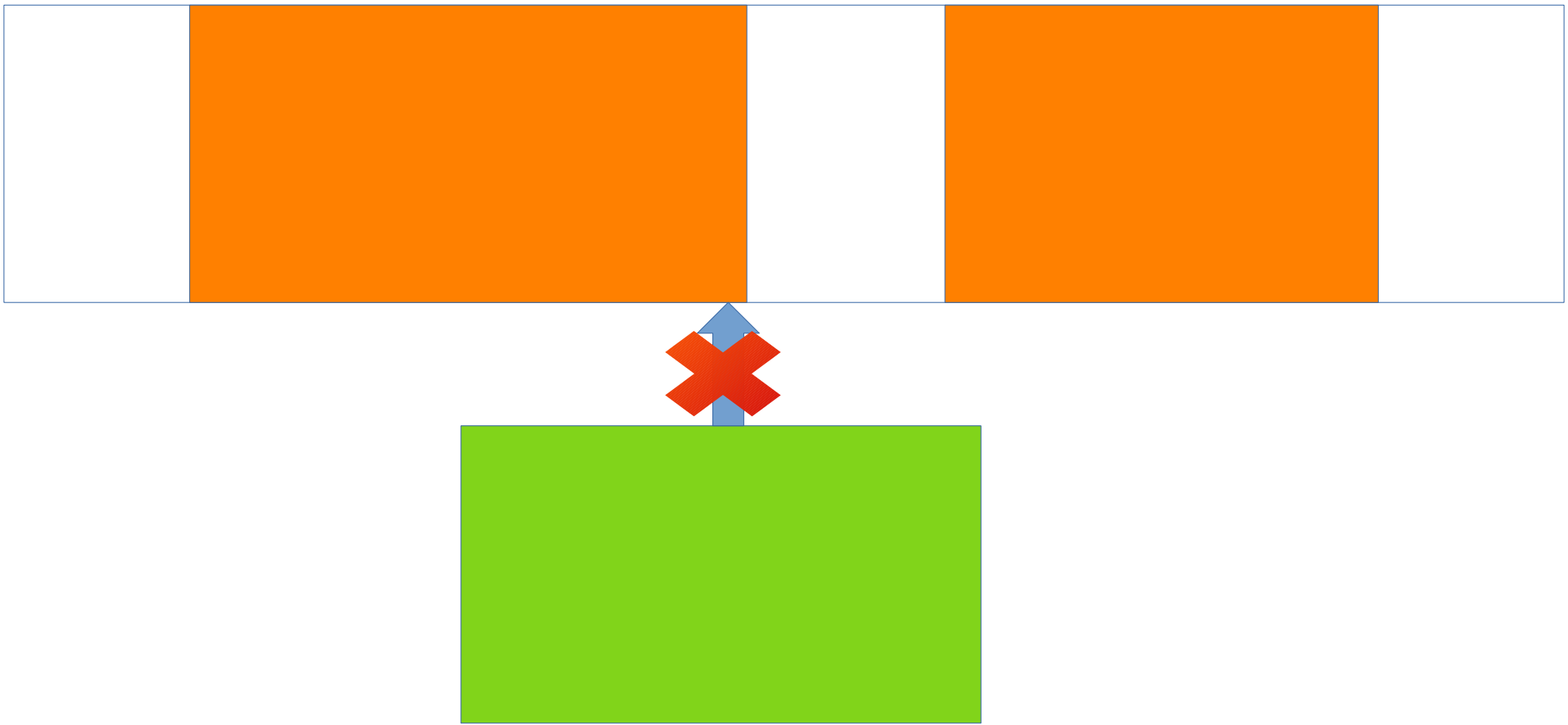
Mémoire

```
auto data_buf = malloc_device<MODEL_DATATYPE>(m.size_n, q);
auto accumulator = malloc_device<MODEL_DATATYPE>(m.nb_workgroup, q);
auto local_accumulator = std::vector<MODEL_DATATYPE>(m.nb_workgroup);

q.submit([&](handler &h)
         { h.memcpy(data_buf, m.data, m.size_n * sizeof(MODEL_DATATYPE)); })
    .wait();

q.submit([&](handler &h)
         { h.memset(accumulator, 0, m.nb_workgroup * sizeof(MODEL_DATATYPE)); })
    .wait();

q.submit([&](handler &h)
         { h.parallel_for(m.nb_workgroup, [=](auto index)
                          {
        size_t id = index[0];
        MODEL_DATATYPE sum = 0;
        for (size_t i = id; i < m.size_n; i+=m.nb_workgroup) {
            sum+=compute(data_buf[i]);
        }
        accumulator[id]=sum; }); });

q.wait();
```

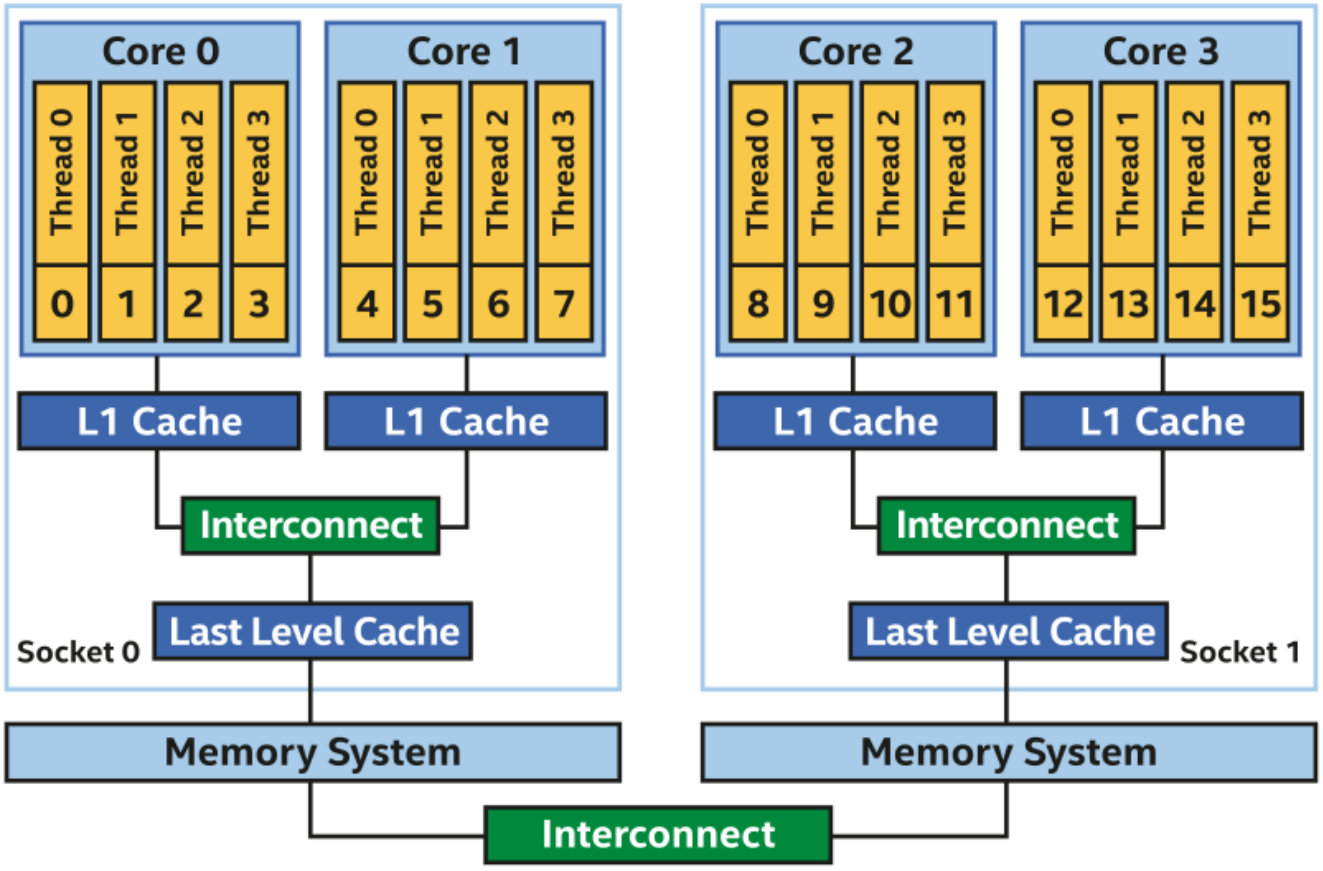
Fragmentation mémoire



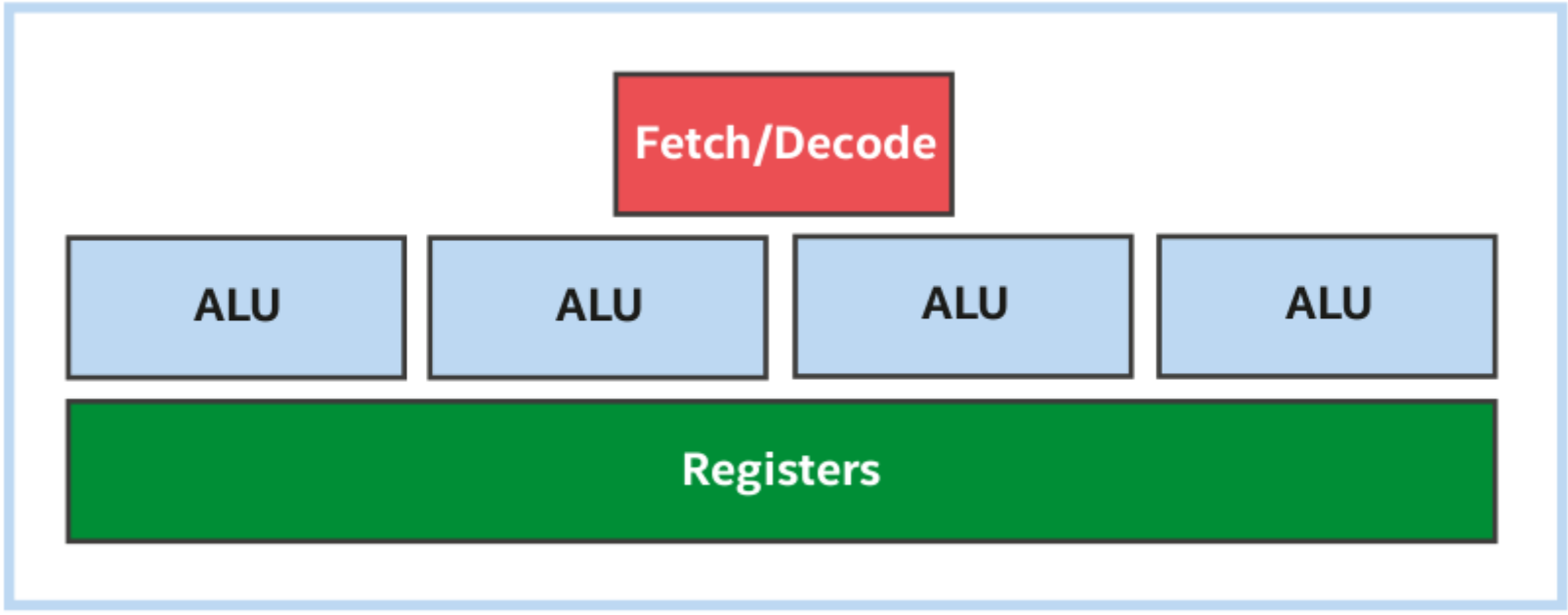
Fragmentation



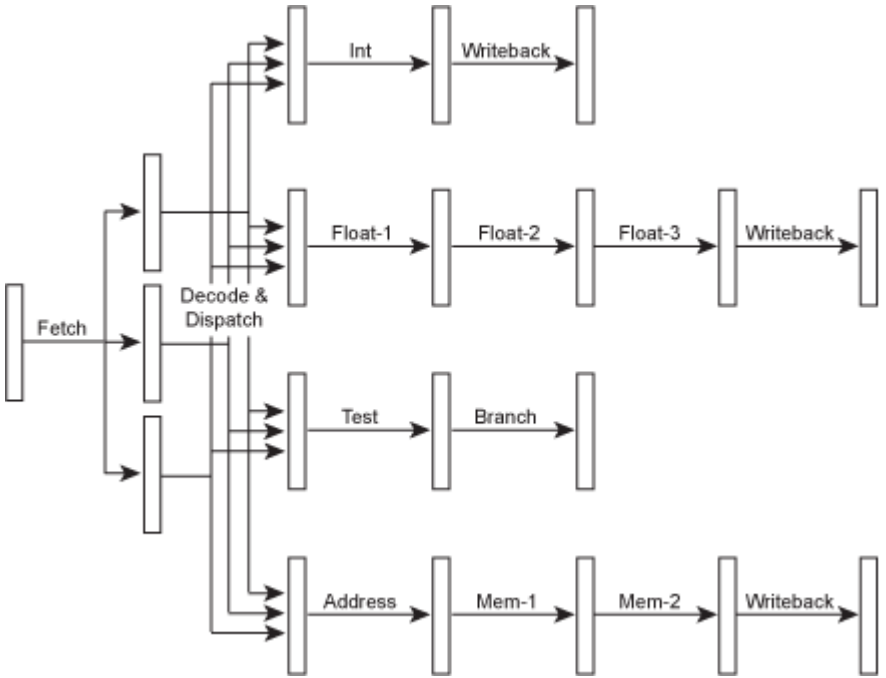
CPU


Core 0
Thread 0
Thread 1
Thread 2
Thread 3
0
1
2
3
Core 1
Thread 0
Thread 1
Thread 2
Thread 3
4
5
6
7
Core 2
Thread 0
Thread 1
Thread 2
Thread 3
8
9
10
11
Core 3
Thread 0
Thread 1
Thread 2
Thread 3
12
13
14
15
L1 Cache
L1 Cache
L1 Cache
L1 Cache
Interconnect
Interconnect
Last Level Cache
Last Level Cache
Socket 0
Socket 1
Memory System
Memory System
Interconnect

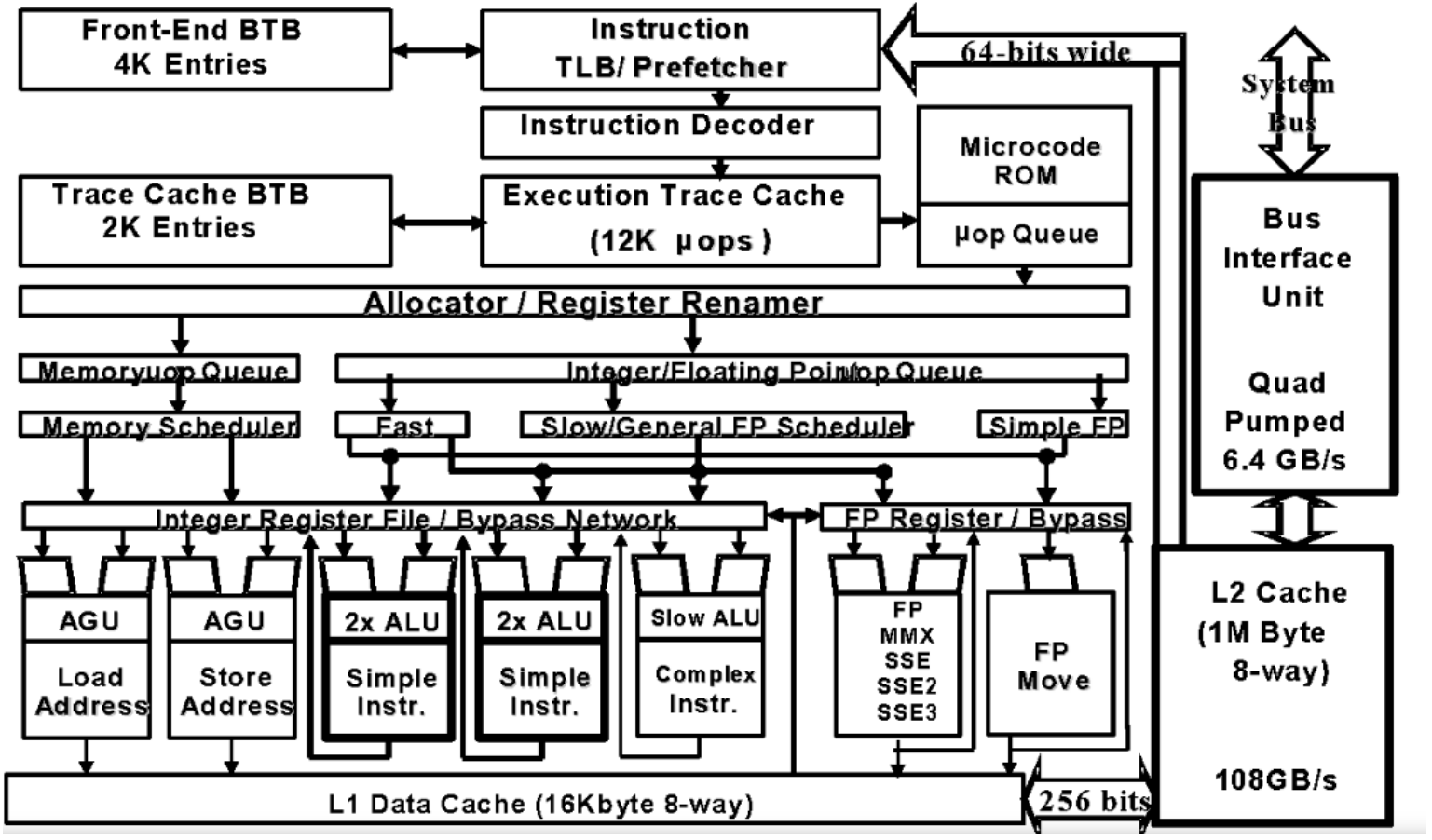
SIMD



PIPELINE


Int
Writeback
Float-1
Float-2
Float-3
Writeback
Fetch
Decode &
Dispatch
Test
Branch
Address
Mem-1
Mem-2
Writeback

SMT


Front-End BTB
4K Entries
Instruction
TLB/ Prefetcher
64-bits wide
System
Bus
Instruction Decoder
Microcode
ROM
Trace Cache BTB
2K Entries
Execution Trace Cache
(12K µops)
µop Queue
Bus
Interface
Unit
Quad
Pumped
6.4 GB/s
Allocator / Register Renamer
Memoryuop Queue
Integer/Floating Point uop Queue
Memory Scheduler
Fast
Slow/General FP Scheduler
Simple FP
Integer Register File / Bypass Network
FP Register / Bypass
AGU
Load
Address
AGU
Store
Address
2x ALU
Simple
Instr.
2x ALU
Simple
Instr.
Slow ALU
Complex
Instr.
FP
MMX
SSE
SSE2
SSE3
FP
Move
L2 Cache
(1M Byte
8-way)
108GB/s
L1 Data Cache (16Kbyte 8-way)
256 bits