SIMD Instructions

Data-level Parallelism – SIMD

- Operate on multiple data with a single instruction
- In this class and project 3: Intel SSE Intrinsics

Intel SSE Intrinsics

- Special 128-bit registers (XXM0-7; 0-15 if x86 64bit architecture) to hold 128-bit SIMD vector data types (eg. __m128 for 4 sp floats, __m128d for 4 doubles).
- Use SSE intrinsics functions to operate on these data types (eg. _mm_add_ps).
- A short example:

```
float A[] = {1, 2, 3, 4, 5, 6, 7, 8}, result[4];
__m128 a1, a2, b;
// a1 = [1, 2, 3, 4]
a1 = _mm_load_ps(A);
// a2 = [5, 6, 7, 8]
a2 = _mm_load_ps(A+4);
// b = [1+5, 2+4, 3+6, 4+8]
b = _mm_add_ps(a1, a2);
// result will be [6, 8, 10, 12]
_mm_store_ps(result, b)
```

MOESI Cache Coherence

State	*Cache up to date?*	*Memory up to date?*	*Others have copy?*	*Can respond to other's reads?*	*Can write without changing state?*
Modified	YES	NO	NO	YES, REQUIRED	YES
Owned	YES	MAYBE	MAYBE	YES, OPTIONAL	NO
Exclusive	YES	YES	NO	YES, OPTIONAL	NO
Shared	YES	MAYBE	MAYBE	NO	NO
Invalid	NO	MAYBE	MAYBE	NO	NO

With the MOESI concurrency protocol implemented, accesses to cache accesses appear *serializiable*. This means that the result of the parallel cache accesses appear the same as if there were done in serial from one processor in *some* ordering.

1. Consider the following access pattern on a two-processor system with a direct-mapped, write-back cache with one cache block and a two cache block memory. Assume the MOESI protocol is used, with write-back caches, and invalidation of other caches on write (instead of updating the value in the other caches).

Time	*After Operation*	*P1 cache state*	*P2 cache state*	*Memory @ 0 up to date?*	*Memory @ 1 up to date?*
0	P1: read block 1	Exclusive (1)	Invalid	YES	YES
1	P2: read block 1				
2	P1: write block 1				
3	P2: write block 1				
4	P1: read block 0				
5	P2: read block 0				

2. Consider if we run the following two loops in parallel (as two threads on two processors).

```
for(int i = 0; i < N; i+=2)  array[i] += 1;
for(int j = 1; j < N; j+=2) array[j] += 2;
```

Would we expect more, less, or the same number of cache misses than if we were to run this serially (assume each processor has its own cache and all data is invalid to start with)?

Concurrency

1. Consider the following function:

```
void transferFunds(struct account *from,
                   struct account *to,
                   int cents) {
      from->cents -= cents;
      to->cents += cents;
}
```

a) What are some data races that could occur if this function called simultaneously from two (or more) threads on the same account? (i.e. if the following code is run)

```
struct account* accounts[9000]; // not all the accounts are unique
#pragma omp parallel for
for (int i = 0; i < 9000; i+=2) {
      transferFunds(accounts[i], accounts[i+1], 5);
}
```

b) How could you fix or avoid these races?