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 64-bit 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:

  ```c
  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)
  ```

- Reference guide: http://software.intel.com/file/18072/

Thread-level Parallelism – OpenMP (Open Multiprocessing)
- An API for multiprocessing programming
- Allows code to be run with multiple threads for faster execution time (multithreading).
- Quick reference: #pragma omp ...
  - parallel – run block/statement in parallel based on configured number of threads.
  - for – make loop index private, run a slice in each thread in parallel
  - critical – only let one thread execute code at a time
  - private(var1, var2, var3 …) – make variables thread-local
  - reduction(operation: variable) – make variable private, combine values of variable in each thread using operation after loop.
  - barrier – threads can’t continue until all threads get to barrier
  - omp_get_thread_num – function (declared in omp.h) to get number of current thread.
Data Synchronization

MOESI Protocol:

<table>
<thead>
<tr>
<th>State</th>
<th>Cache up to date?</th>
<th>Memory up to date?</th>
<th>Others have copy?</th>
<th>Can respond to other's reads?</th>
<th>Can write without changing state?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES, REQUIRED</td>
<td>YES</td>
</tr>
<tr>
<td>Owned</td>
<td>YES</td>
<td>NO</td>
<td>MAYBE</td>
<td>YES, REQUIRED</td>
<td>YES</td>
</tr>
<tr>
<td>Exclusive</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>YES, OPTIONAL</td>
<td>NO</td>
</tr>
<tr>
<td>Shared</td>
<td>YES</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Invalid</td>
<td>NO</td>
<td>MAYBE</td>
<td>MAYBE</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

- Why do we care about this? Keeping data synchronized across modern multiprocessor platform is an important topic! We need to have some communication protocol across all processors.

- With the MOESI concurrency protocol implemented, accesses to cache appear *serializable*. 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.

- False sharing: Occurs when two processors access data elements that happen to be in the same cache block. The cache block is constantly marked *modified* so each processor has to constantly reload the cache block even though it is unnecessary.

- Data race: Happens when two processes load the same data element and one’s result overwrites the other’s in the memory (classic bank account example: balance only reflects one transaction’s effect when two transactions happen at the same time)

Summary of general speed-up techniques (might come in handy for project 3)

- *Data-level parallelism / SIMD*: compute multiple results at a time.
- *Thread-level parallelism / OpenMP*: have multiple threads doing computations at a time
- *I/D cache locality (ie. loop ordering, etc.)*: maximize cache hits for higher speed.
- *Loop unrolling*: minimizes for loop overheads.
- *Cache blocking*: increase cache usage for higher performance.
- *Code optimization (mostly compiler’s job)*: interweave independent instructions to avoid CPU stalls (waiting for the results from the previous instruction).