

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

- 1.1 SIMD is ideal for flow-control heavy tasks (i.e. tasks with many branches/if statements).
- 1.2 Intel's SIMD intrinsic instructions invoke large registers available on the architecture in order to perform one operation on multiple values at once.
- 1.3 The pipelined datapath is an example of parallelism because it performs different stages of instructions in parallel.
- 1.4 The most effective way of increasing performance on a modern PC is to increase its clock speed.
- 1.5 In thread-level parallelism, the amount of speedup is directly proportional to the increase in number of cores.
- 1.6 In thread-level parallelism, threads may run in any order and can start while other threads are partway through their execution.

2 Write-Back Caches

When it comes to writing data to cache memory, there are multiple write policies to consider that offer different options when building our system. Some of them you might encounter are:

1. **Write-through:** In this policy, when we have a write we write to both the cache and the memory. This is the case for every write, so the main memory always has the updated data. This is simple to implement, but writing to main memory every single time is slow.
2. **Write-back:** On a write, the data is only updated/written in the cache. The main memory only receives the data upon eviction. This means the cache has more up to date data most of the time. While this is faster as there is less accesses to main memory, it is harder to implement as we have to include more overhead, such as dirty bits and so on.

2.1 Considering the above information, lets consider a direct mapped cache with a capacity of 16B and a block size of 4B. Lets also assume that the memory addresses are 8 bits each. Assuming the cache is completely empty in the beginning, we make memory accesses to the following locations:

- 0x6A, Write
- 0x0F, Write
- 0x38, Read
- 0x6B, Read
- 0x81, Read
- 0x87, Write
- 0x68, Write
- 0x6B, Read

Fill out the metadata of the cache if we use a write-back policy.

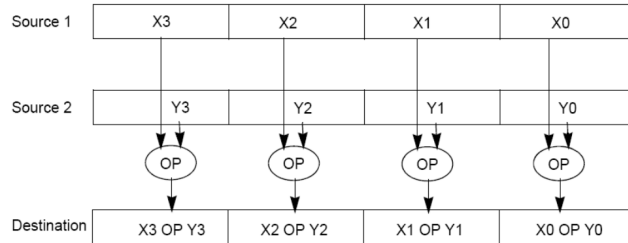
Index	Valid	Dirty	Tag
0			
1			
2			
3			

2.2 How many times will we write to main memory in this case?

2.3 Now suppose we had a write-through policy. How many times will we write to main memory in this case?

3 Data-Level Parallelism

The idea central to data level parallelism is vectorized calculation: applying operations to multiple items (which are part of a single vector) at the same time.



Some machines with x86 architectures have special, wider registers, that can hold 128, 256, or even 512 bits. Intel intrinsics (Intel proprietary technology) allow us to use these wider registers to harness the power of DLP in C code.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. The type `__m128i` is used when these registers hold 4 ints, 8 shorts or 16 chars; `__m128d` is used for 2 double precision floats, and `__m128` is used for 4 single precision floats. Where you see “epiXX”, epi stands for **e**xtended **p**acked **i**nteger, and XX is the number of bits in the integer. “epi32” for example indicates that we are treating the 128-bit register as a pack of 4 32-bit integers.

- `__m128i _mm_set1_epi32(int i)`:
Set the four signed 32-bit integers within the vector to `i`.
- `__m128i _mm_loadu_si128(__m128i *p)`:
Load the 4 successive ints pointed to by `p` into a 128-bit vector.
- `__m128i _mm_mullo_epi32(__m128i a, __m128i b)`:
Return vector $(a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)$.
- `__m128i _mm_add_epi32(__m128i a, __m128i b)`:
Return vector $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- `void _mm_storeu_si128(__m128i *p, __m128i a)`:
Store 128-bit vector `a` at pointer `p`.
- `__m128i _mm_and_si128(__m128i a, __m128i b)`:
Perform a bitwise AND of 128 bits in `a` and `b`, and return the result.
- `__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)`:
The `i`th element of the return vector will be set to `0xFFFFFFFF` if the `i`th elements of `a` and `b` are equal, otherwise it'll be set to 0.

- 3.1 SIMD-ize the following function, which returns the product of all of the elements in an array.

```
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product *= a[i];
    }
    return product;
}
```

Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration? What if our array has a length that isn't a multiple of 4? What can we do to handle this tail case?

```
static int product_vectorized(int n, int *a) {
    int result[4];
    __m128i prod_v = _____;
    for (int i = 0; i < ____; i += __) { // Vectorized loop
        prod_v = _____;
    }
    __mm_storeu_si128(_____, _____);
    for (int i = _____; i < _____; i++) { // Handle tail case
        result[0] *= _____;
    }
    return _____;
}
```

4 Thread-Level Parallelism

OpenMP provides an easy interface for using multithreading within C programs. Some examples of OpenMP directives:

- The `parallel` directive indicates that each thread should run a copy of the code within the block. If a for loop is put within the block, **every** thread will run every iteration of the for loop.

```
#pragma omp parallel
{
    ...
}
```

NOTE: The opening curly brace needs to be on a newline or else there will be a compile-time error!

- The `parallel for` directive will split up iterations of a for loop over various threads. Every thread will run **different** iterations of the for loop. The exact order of execution across all threads, as well as the number of iterations each thread performs, are both non-deterministic, as the OpenMP library load balances threads for performance. The following two code snippets are equivalent.

```
#pragma omp parallel for          #pragma omp parallel
for (int i = 0; i < n; i++) {    {
    ...                          #pragma omp for
                                for (int i =0; i < n; i++) { ... }
}                                  }
```

There are two functions you can call that may be useful to you:

- `int omp_get_thread_num()` will return the number of the thread executing the code
- `int omp_get_num_threads()` will return the number of total hardware threads executing the code

4.1 For each question below, state and justify whether the program is **sometimes incorrect, always incorrect, slower than serial, faster than serial, or none of the above**. Assume the number of threads can be any integer greater than 1. Assume `arr` is an `int[]` of length `n`.

(a) // Set element `i` of `arr` to `i`

```
#pragma omp parallel
{
    for (int i = 0; i < n; i++)
        arr[i] = i;
}
```

(b) // Set `arr` to be an array of Fibonacci numbers.
`arr[0] = 0;`

```
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
    arr[i] = arr[i-1] + arr[i - 2];
```

(c) // Set all elements in arr to 0;

```
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    arr[i] = i;
```

(d) // Set element i of arr to i;

```
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    *arr = i;
    arr++;
```

5 Amdahl's Law

In attempting to parallelize a program, the overall performance speedup will always be limited by the fraction of the program that cannot be sped up. This overall speedup can be formulated by Amdahl's Law, which states that

$$\mathbf{Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}$$

Speedup refers to the theoretical speedup of the program compared to its naive implementation. Note that **Speedup** $>$ 1 since we're making our program faster than the original.

F refers to the fraction of the program that can be optimized;

S is the speedup factor for how much that portion of the program can be optimized by, where (**S** $>$ 1)

5.1 Derive Amdahl's Law using the ratio: **Speedup** = $t_{\text{naive}}/t_{\text{optimized}}$

5.2 Assuming we have infinite threads and resources, what would our overall speedup be for a program with some fraction of our code that can be parallelized F ?
rin

5.3 You write code that will search for the phrases "Hello Sean", "Hello Jon", "Hello Dan", "Hello Man", "Bora is the Best!" in text files. With some analysis, you determine you can speed up 40% of the execution by a factor of 2 when parallelizing your code. What is the true speedup?

5.4 You run a profiling program on a different program to find out what percent of time within the program each function takes. You get the following results:

Function	% Time
f	30%
g	10%
h	60%

- (a) Assuming that each of these functions can be parallelized by the same speedup factor, which one, if parallelized, would cause the most speedup for the entire program?
- (b) What speedup would you get if you parallelized just this function with 8 threads? Assume that work is distributed evenly across threads and there is no overhead for parallelization.