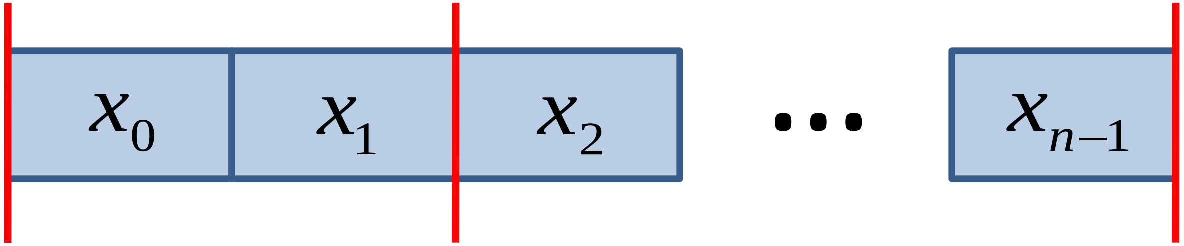
Fast Vectorization with Compiler Intrinsics

…. The Memory-Alignment Problem

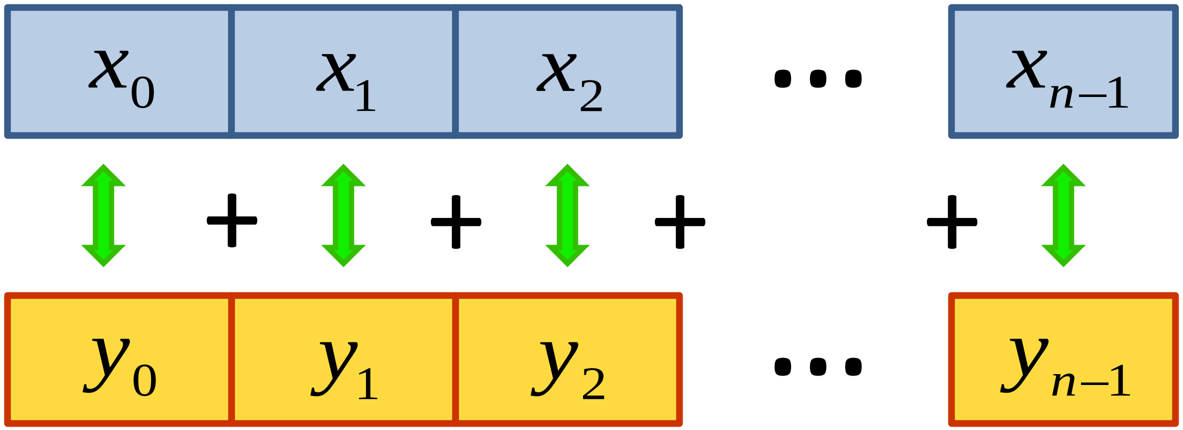


Graham Beck

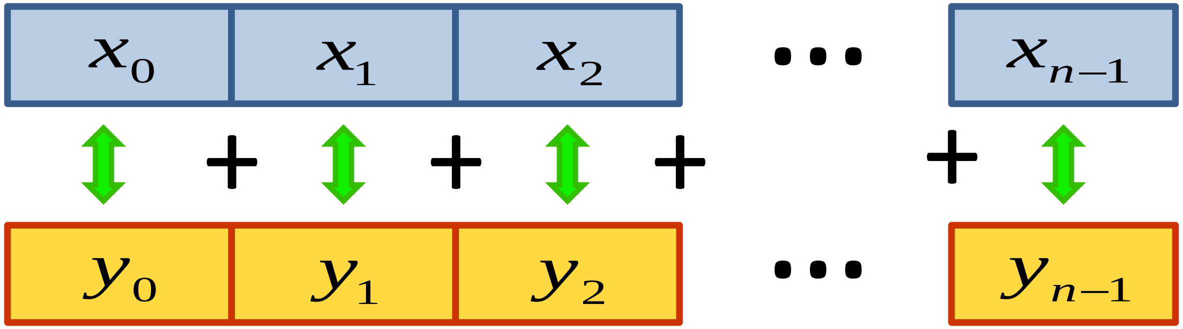
Sept 17, 2015

ALLSTON TRADING

Objective: Fast calculation of the **inner product** $\sum_i x_i y_i$



for **contiguous-memory** arrays



Basic Form: Simple For-loop

```
for (i=0; i < n; ++i) {
      sum += x[i]*y[i];
}
```

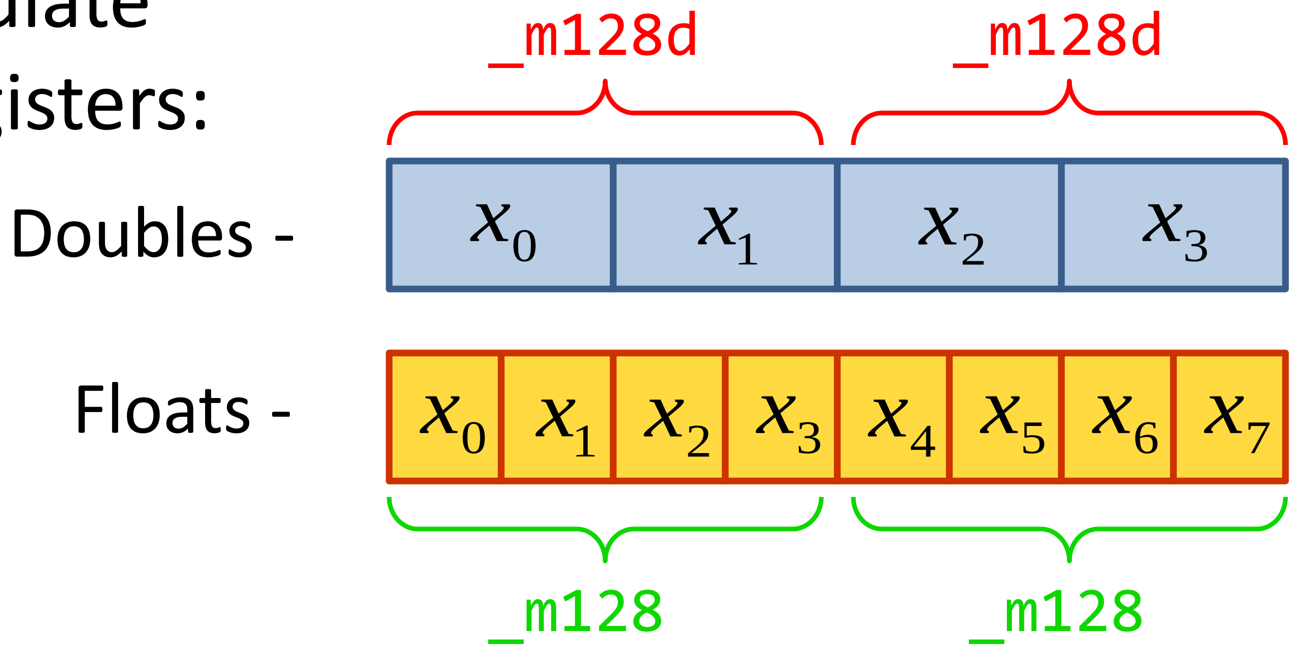
Intermediate: Loop unrolling

```
for (i=0; i < n; i+=2) {
      sum1 += x[i]*y[i];
      sum2 += x[i+1]*y[i+1];
}
```

Advanced: Explicit use of compiler intrinsics

- Intrinsics are **built-in functions**
- Optimized processor-specific machine instructions
- Often faster than inline assembly

We manipulate 128-bit registers:



Tools: `double* const x, y; union {double[2] d; _m128d ip;} u;`

Loading:

```
_m128d xr=_mm_load_pd(x)
_m128d yr=_mm_load_pd(y)
```

xr :	x_0	x_1

Dot Product:

```
_m128d dp01=_mm_dp_pd(xr,yr,mask)
```

dp01 :	$x_0 \times y_0$	$x_1 \times y_1$

Horizontal Add:

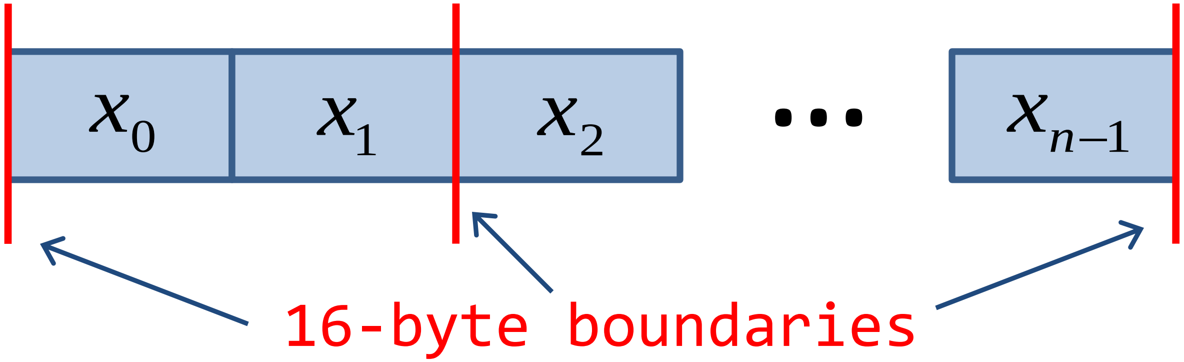
```
u.ip=_mm_hadd_pd(dp01,dp23)
```

u.ip :	$x_0 \times y_0 + x_1 \times y_1$	$x_2 \times y_2 + x_3 \times y_3$

```
sum += u.d[0] + u.d[1]
```

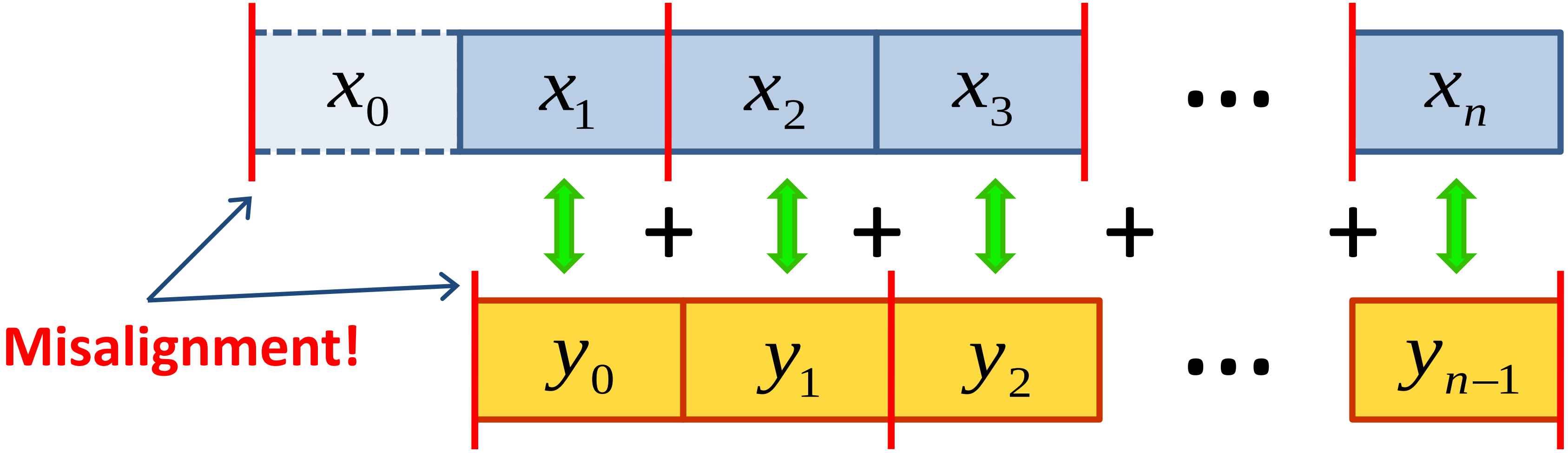
So we have a fast way of taking the inner product

…but the key to this is **16-byte aligned memory**



16-byte boundaries

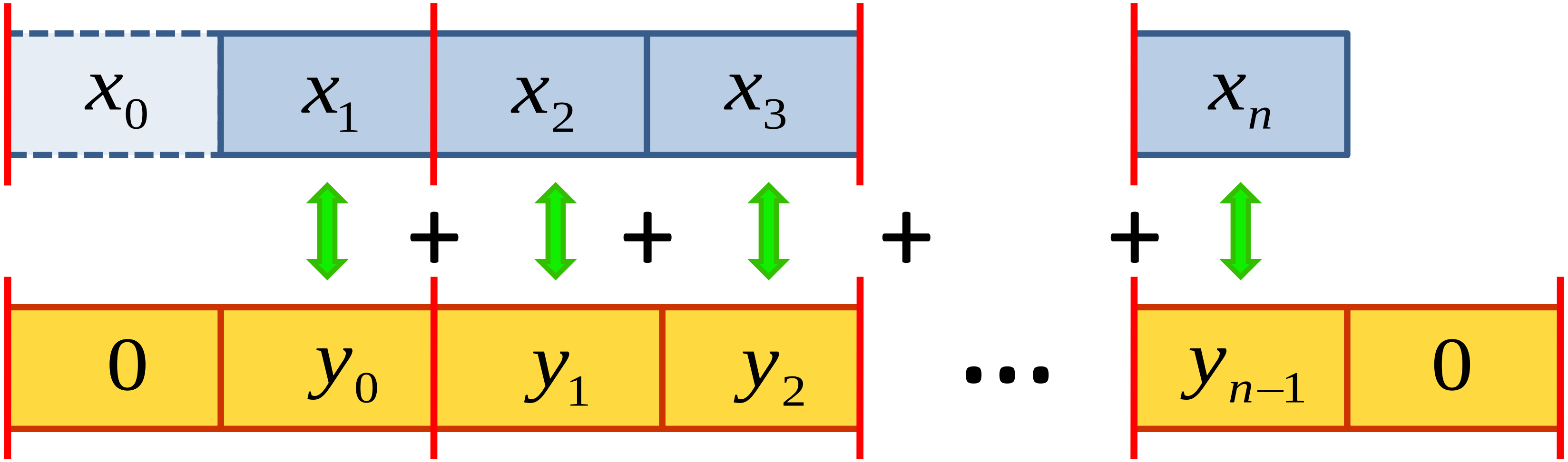
But what happens when we roll forward?



Misalignment!

Either take performance hit with slower, generic instructions `_m128d xr=_mm_load_sd(x)` etc …

Or match the new alignment of x with a new y



So the array y alternates (over two for doubles, four for floats) based on x's alignment.