Compilers for Embedded Systems

Integrated Systems of Hardware and Software

Lecture 4

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk
Website: https://www.plymouth.ac.uk/staff/vasilios-kelefouras

School of Computing
(University of Plymouth)

Outline

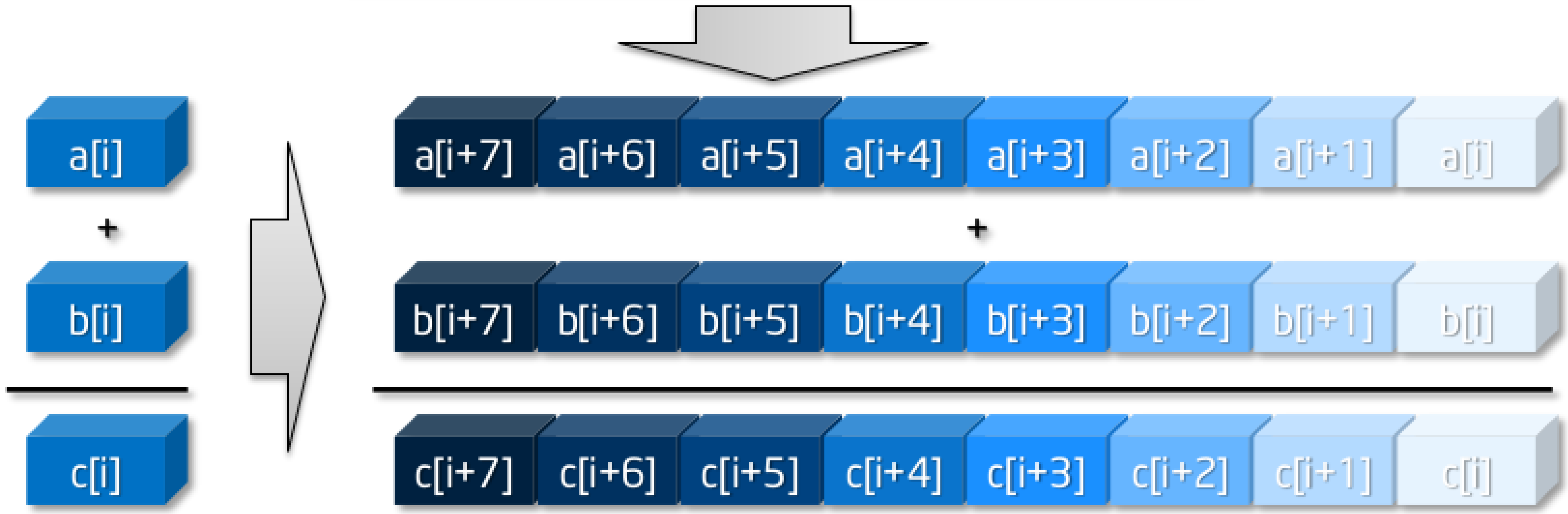
- Different ways of using vectorization
- Using intrinsic functions in C/C++
- Writing C/C++ programs using x86-64 SSE intrinsics
- Writing C/C++ programs using x86-64 AVX intrinsics

Introduction

- All modern processors support vectorization.
- This means that **processors have extra hardware components** (wide registers and wide processing units) to allow vector processing.
- Vectorization is the process of processing vectors (multiple values together) instead of single values.
- It is also known as Single Instruction Multiple Data (SIMD), as a single instruction is used to process multiple data.
- Vectorization **dramatically improves the performance of our code**.
- The compilers apply vectorization automatically, this process is called *auto-vectorization*, but not always with success.
- Vectorization has nothing to do with the language used.

Single Instruction Multiple Data (SIMD) – Vectorization

```
for(i = 0; i <= MAX;i++)
    c[i] = a[i] + b[i];
```


a[i]
+
b[i]
c[i]
a[i+7] a[i+6] a[i+5] a[i+4] a[i+3] a[i+2] a[i+1] a[i]
+
b[i+7] b[i+6] b[i+5] b[i+4] b[i+3] b[i+2] b[i+1] b[i]
c[i+7] c[i+6] c[i+5] c[i+4] c[i+3] c[i+2] c[i+1] c[i]

Vectorization can be applied in four main ways

1. **Automatically by the compiler**
   - (Linux) specify '-O3' option or *'-ftree-slp-vectorize'* , *'-ftree-vectorize*'.
   - (Linux) Compile using *'-fopt-info-vec-optimized*' option to see which parts are vectorized.
   - (VS) Use the Release mode
   - This solution does not provide the best performance, but it is very easy to use.
2. **Using OpenMP C/C++ pragmas.**
   - Better performance than above, less easy to use (this will be studied later on).
3. **Using C/C++ instrinsics (assembly coded functions).**
   - Even Better performance (close to assembly), not that easy to use.
4. **Directly writing assembly code** (not used in this module).
   - Best performance but hard to use.

Auto-Vectorization

- '-O3' will auto-vectorize your code (at least gcc will try).
  - Example:

gcc main.cpp array_addition.cpp array_constant_addition.cpp MVM.cpp -o p -march=native -O3 -fopt-info-vec-optimized

- '*-fopt-info-vec-optimized*' option to see which parts are vectorized

- If '-O3' is **not** included, you can enable vectorization manually by using

  -fopt-info-vec-optimized -ftree-vectorize
  - Example:

gcc main.cpp array_addition.cpp array_constant_addition.cpp MVM.cpp -o p -march=native -O2 -fopt-info-vec-optimized -ftree-vectorize

Vectorization using OpenMP 4.0

- There is a large number of clauses supported by OpenMP such as reduction, simdlen etc, to allow efficient and easy vectorization even in cases where the loop kernel is complex.
- OpenMP will not be studied this week.

```
#pragma omp simd
For (int n=0; n<N; ++n)
   a[n] += b[n];    //the compiler will vectorize this if possible


#pragma omp simd aligned(a,b:16) safelen(4)
for (int i=4; i<N; ++i)
   a[i] = a[i-4] * 2;
```

Using intrinsic functions in C/C++

- **Main advantages**
  - Portability to almost all x86 architectures
  - Compatibility with different compilers

- **Main disadvantages**
  - Not all assembly instructions have intrinsic function equivalents
  - Unskilled use of intrinsic functions can make the code less efficient than simple C++ code

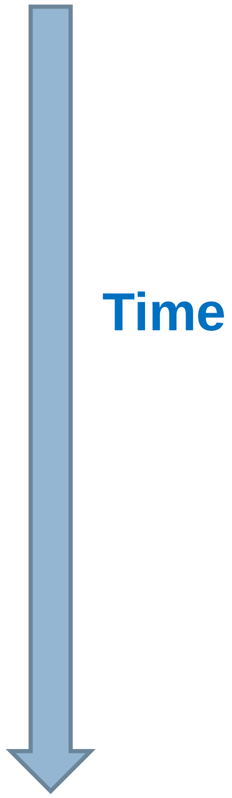
Using intrinsic functions in C/C++

- **For the rest of this lecture, you will be learning how to use intrinsic functions in C/C++**

- Normally, "90% of a program's execution time is spent in executing 10% of the code" - **loops**
  - What programmers normally do to improve performance is to analyze the code and find the computationally intensive functions
    - Then optimize those instead of the whole program
    - This safes time and money
  - **Rewriting loop kernels in C++ using SIMD intrinsics is an excellent choice**
    - Compilers vectorize the code (not always), but doing that manually, using SIMD instrinsics, can really boost performance

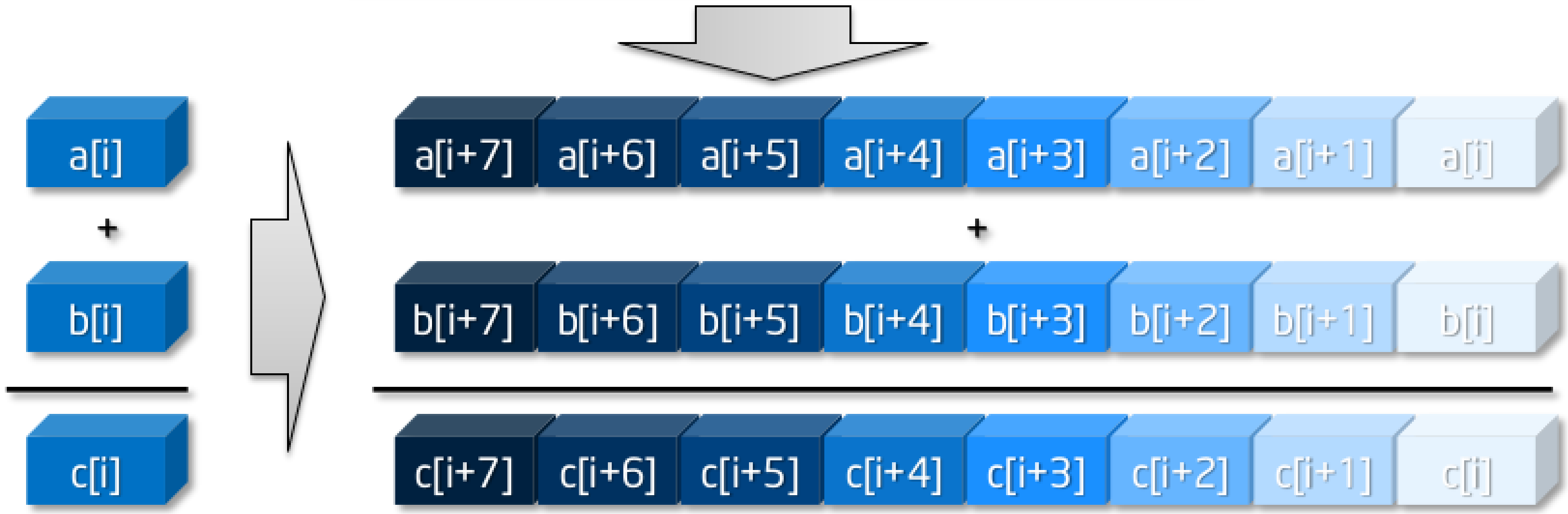
Computer Systems - Hardware Evolution

- Scalar Processors
- Pipelined Processors
- Superscalar and VLIW Processors
- Out of order Processors
- Processors support **Vectorization**
- Hyperthreading
- Multicore Processors
- Manycore Processors
- Heterogeneous systems



Single Instruction Multiple Data (SIMD) – Vectorization

```
for(i = 0; i <= MAX;i++)
    c[i] = a[i] + b[i];
```

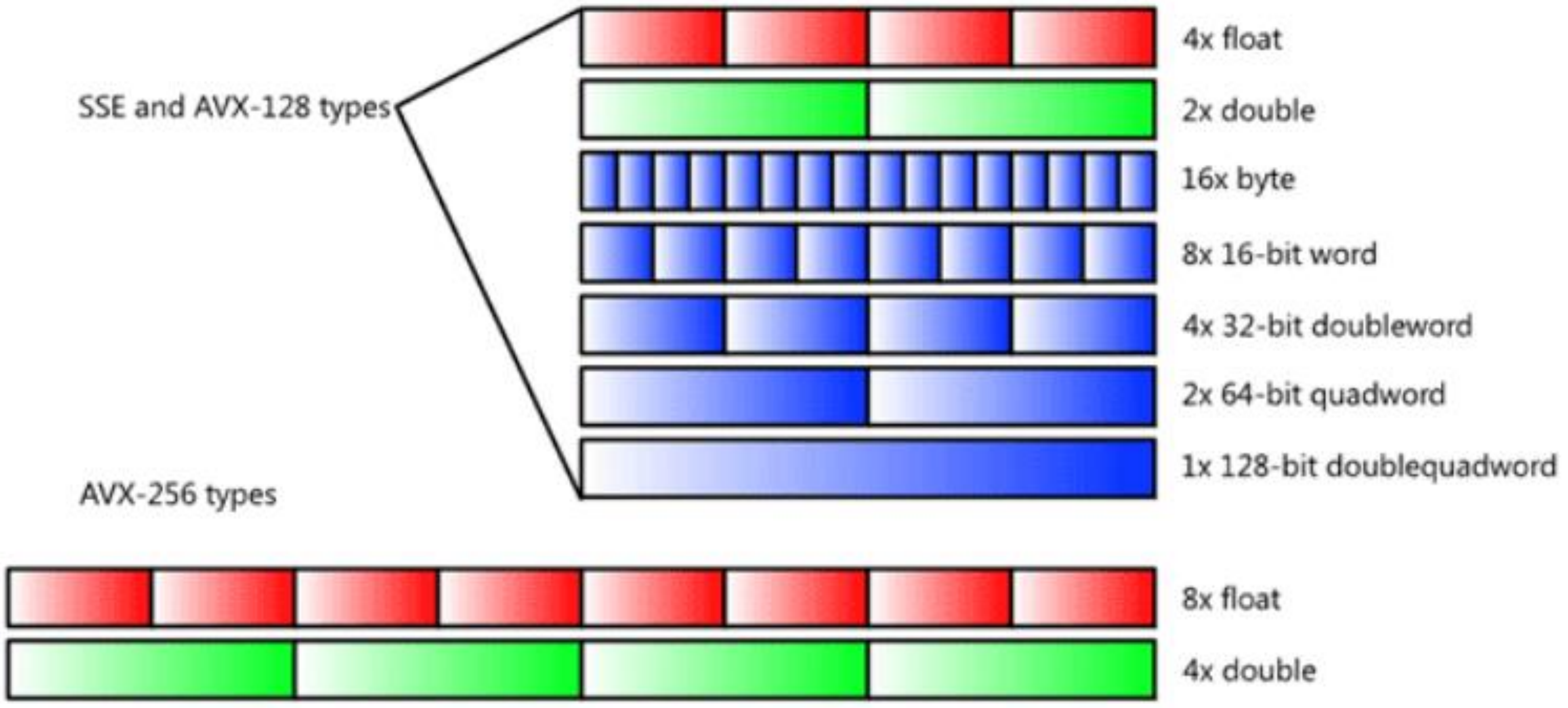

a[i]
+
b[i]
c[i]
a[i+7] a[i+6] a[i+5] a[i+4] a[i+3] a[i+2] a[i+1] a[i]
+
b[i+7] b[i+6] b[i+5] b[i+4] b[i+3] b[i+2] b[i+1] b[i]
c[i+7] c[i+6] c[i+5] c[i+4] c[i+3] c[i+2] c[i+1] c[i]

Vectorization on x86-64 Processors

- **Intel MMX technology** (old – limited usage nowadays)
  - 8 mmx registers of 64 bit
  - extension of the floating point registers
  - can be handled as 8 8-bit, 4 16-bit, 2 32-bit and 1 64-bit, operations
- **Intel SSE technology**
  - 8/16 xmm registers of 128 bit (32-bit architectures support 8 registers only)
  - Can be handled from 16 8-bit to 2 64-bit operations
- **Intel AVX technology**
  - 8/16 ymm registers of 256 bit (32-bit architectures support 8 registers only)
  - Can be handled from 32 8-bit to 4 64-bit operations
- **Intel AVX-512 technology**
  - 32 ZMM 512-bit registers

Vectorization on x86-64 Processors (2)

__m256 or __m128 (for single precision floats)
__m256d or __m128d (for double precision floats)
__m256i or __m128i (for integers, no matter the size)



Vectorization on x86-64 Processors (3)

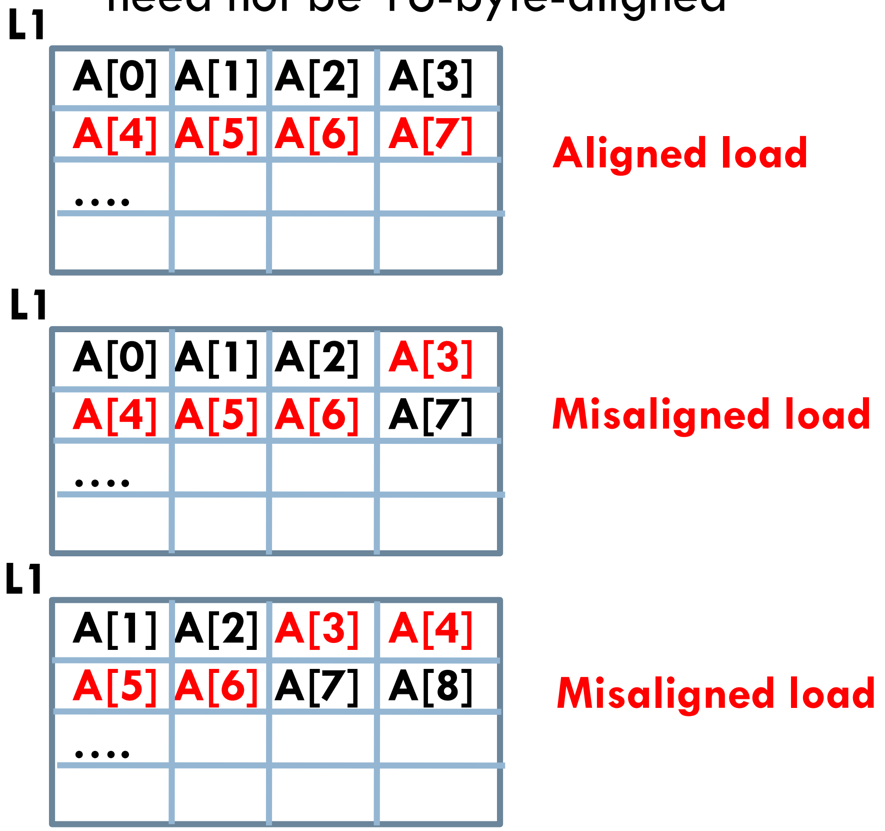
- The developer can use either SSE or AVX or both
  - AVX instructions improve throughput
  - SSE instructions are preferred for less data parallel algorithms
- Most of the load/store vector instructions work only for data that they are written in consecutive main memory addresses
- Normally, aligned load/store instructions are faster than the no aligned ones.
- Memory and arithmetical instructions are executed in parallel

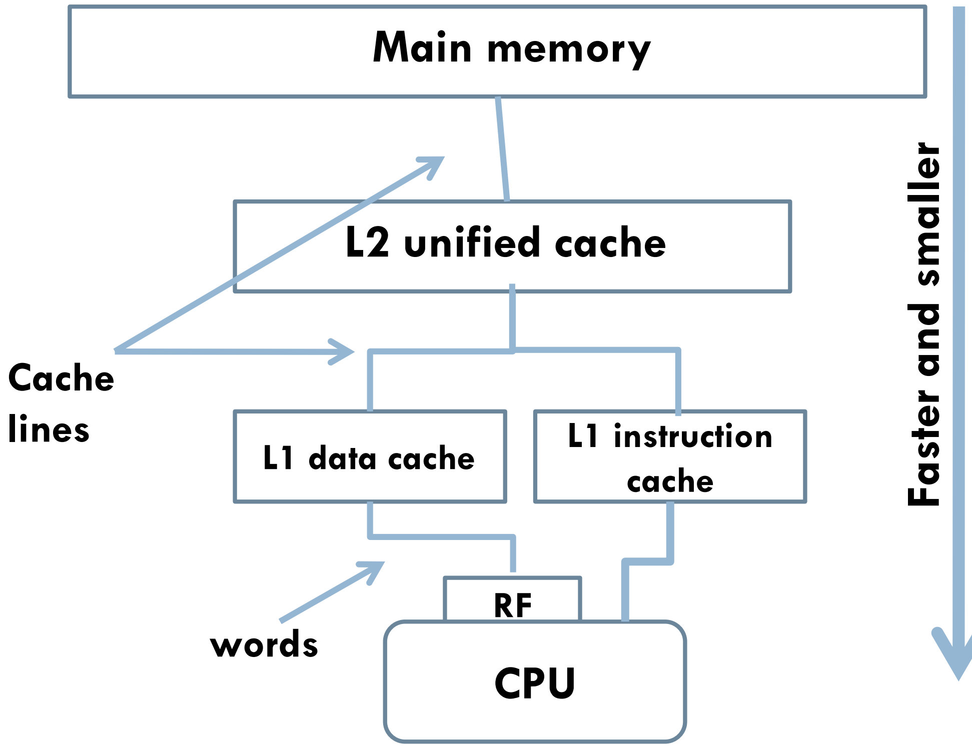
- **All the Intel intrinsics are found here :**

https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

Basic SSE Instructions (1)

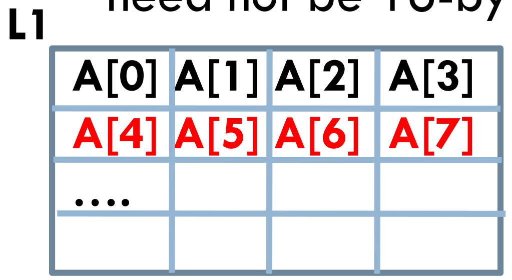
- __m128 **_mm_load_ps** (float * p) – Loads four SP FP values. The address must be 16-byte-aligned, e.g., *var1 = _mm_load_ps (&A[3][4]);*
- __m128 **_mm_loadu_ps** (float * p) - Loads four SP FP values. The address need not be 16-byte-aligned



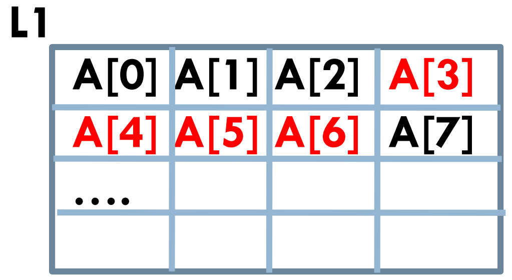


Basic SSE Instructions (2)

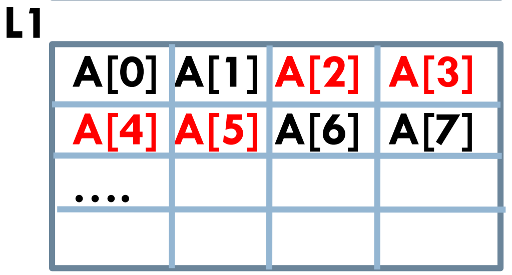
- __m128 **_mm_load_ps**(float * p) – Loads four SP FP values. The address must be 16-byte-aligned
- __m128 **_mm_loadu_ps**(float * p) - Loads four SP FP values. The address need not be 16-byte-aligned



Aligned load

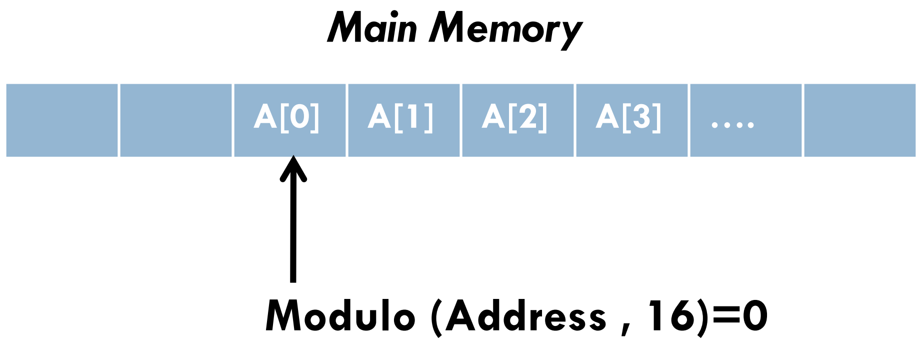


Misaligned load



Misaligned load

__declspec (align(16)) float A[N][N];



Do not forget to align your arrays

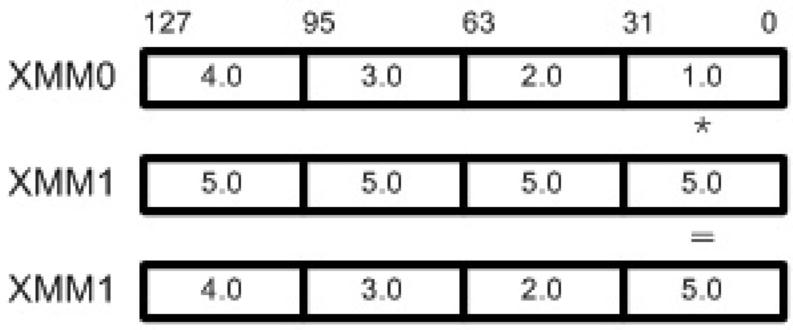
- We can either allocate aligned memory ***statically*** using
  - *float A[N] __attribute__((aligned (64))); //In Linux only*
  - *__declspec(align(64)) float A[N] //In Visual studio only*

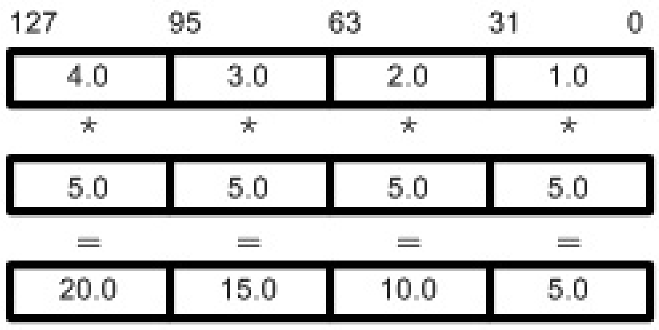
- or ***dynamically*** using
  - *_mm_malloc (N * sizeof(float),64); //Linux and Visual Studio*

Basic SSE Instructions (3)

- __m128 **_mm_store_ps**(float * p) – Stores four SP FP values. The address must be 16-byte-aligned
- __m128 **_mm_storeu_ps**(float * p) – Stores four SP FP values. The address need **not** to be 16-byte-aligned
- __m128 **_mm_mul_ps**(__m128 a, __m128 b) - Multiplies the four SP FP values of a and b
- __m128 **_mm_mul_ss**(__m128 a, __m128 b) - Multiplies the lower SP FP values of a and b; the upper 3 SP FP values are passed through from a.

XMM1=_mm_mul_ss(XMM1, XMM0) **XMM1=_mm_mul_ps(XMM1,XMM0)**

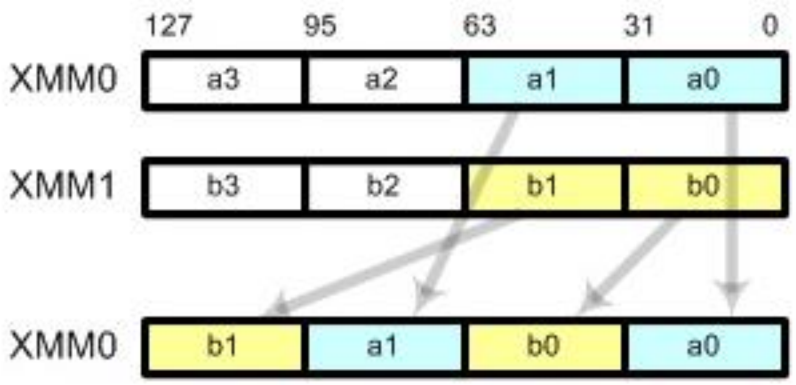




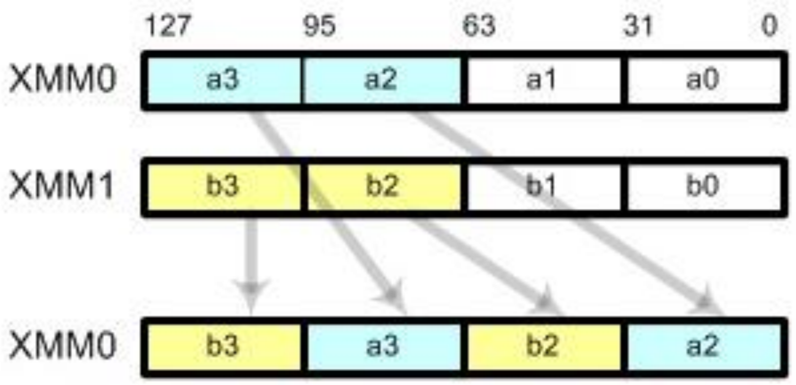
Basic SSE Instructions (4)

- __m128 **_mm_unpackhi_ps** (__m128 a, __m128 b) - Selects and interleaves the upper two SP FP values from a and b.
- __m128 **_mm_unpacklo_ps** (__m128 a, __m128 b) - Selects and interleaves the lower two SP FP values from a and b.

XMM0=_mm_unpacklo_ps (XMMO, XMM1)

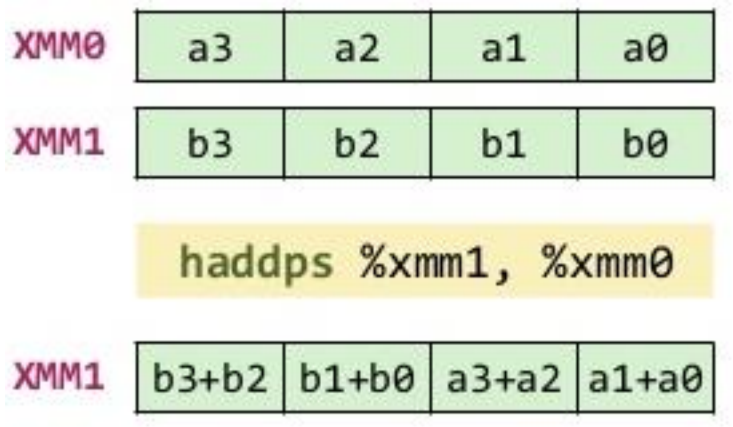


XMM0=_mm_unpackhi_ps (XMMO, XMM1)



Basic SSE Instructions (5)

- **__m128 _mm_hadd_ps** (__m128 a, __m128 b) - Adds adjacent vector elements

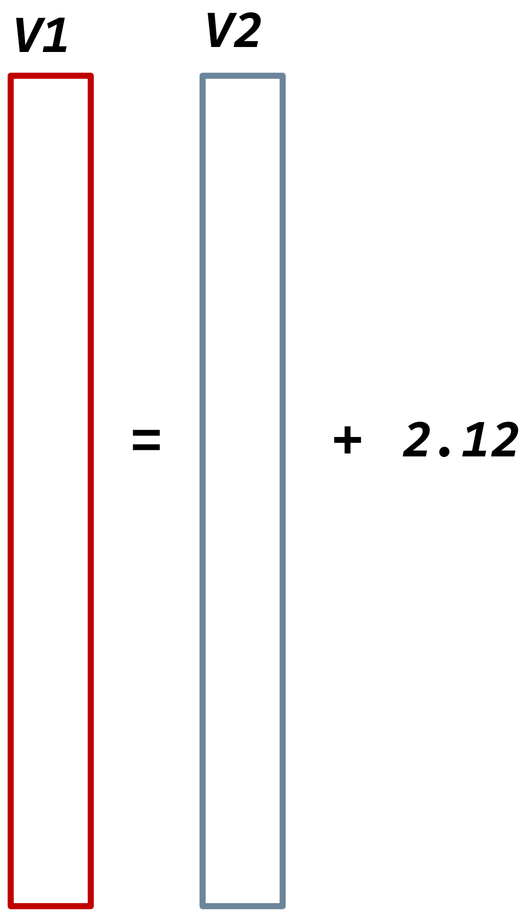


- void **_mm_store_ss** (float * p, __m128 a) - Stores the lower SP FP value

Vectorization 1st example (1)

- Compilers vectorise such simple loop kernels, automatically (auto-vectorization)
- **Without** vectorization
  - Load V2[0]
  - V2[0]+2.12
  - Store the result into V1[0]
- **With** vectorization
  - Load V2[0:3] (load 4 values together)
  - V2[0:3] + 2.12 (apply 4 additions together)
  - Store the result into V1[0:3] (store 4 values together)

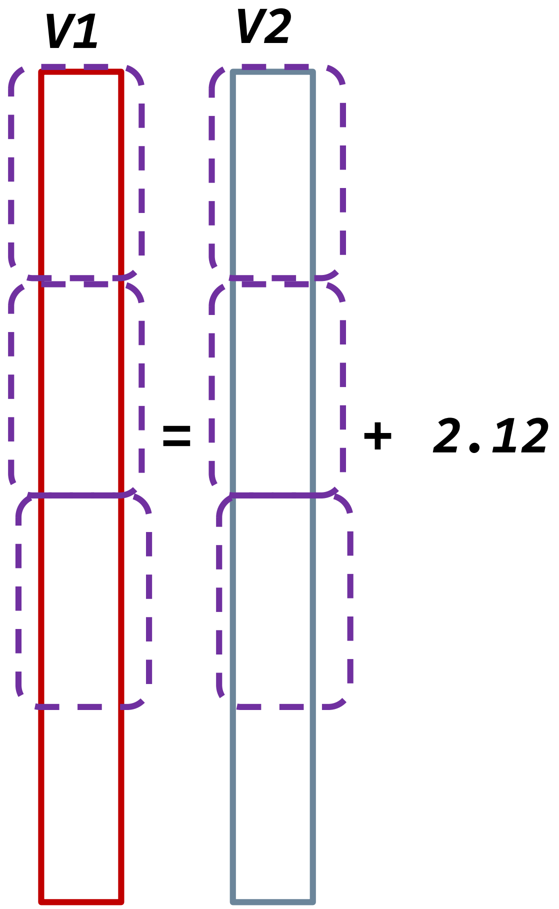
```
for (j = 0; j < M; j++) {
V1[j] = V2[j] + 2.1234;
}
```



Vectorization 1st example (2)

- 4 elements are processed together using vector instructions
- Performance is improved by x4
- Low-level C code is easier than assembly, isn't it?

```
for (j = 0; j < M; j++) {
V1[j] = V2[j] + 2.12;
}
```



```
__m128 num1, num2, num3;

num1 = _mm_set_ps (2.12, 2.12, 2.12, 2.12);

for (int i = 0; i < M; i += 4) {
   num2 = _mm_load_ps (&V2[i]);
   num3 = _mm_add_ps (num1, num2);
   _mm_store_ps( &V1[i], num3);
}
```

Vectorization 1st example (3)

```
for (j = 0; j < M; j++) {
V1[j] = V2[j] + 2.1234;
}
```

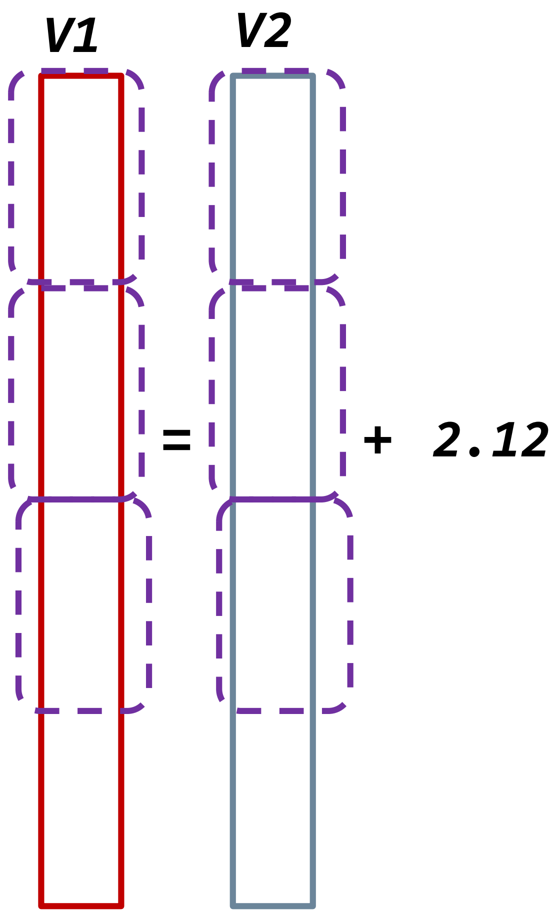
Define three 128bit variables of type float, thus each 128bit variable contains 4 FP values

Initialize the 128bit variable

Load 128bit of data starting from the memory address &V2[j]

4 iterations are processed in every iteration

```
 __m128 num1, num2, num3;

 num1 = _mm_set_ps (2.12, 2.12, 2.12, 2.12);

 for (int i = 0; i < M; i += 4) {
   num2 = _mm_load_ps (&V2[i]);
   num3 = _mm_add_ps (num1, num2);
   _mm_store_ps( &V1[i], num3);
 }
```



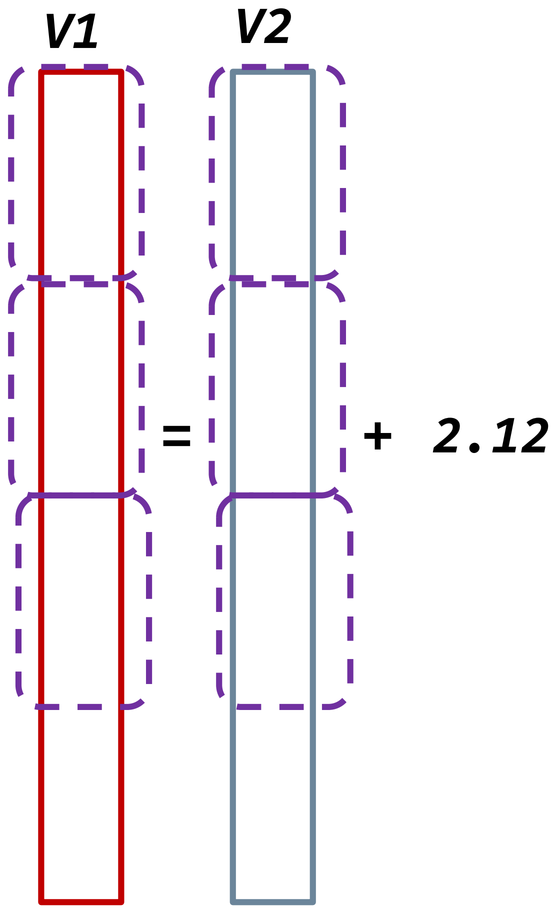
Vectorization 1st example (4)

- ***What if M==10?***
  - Or not a multiple of 4

```
for (j = 0; j < M; j++) {
V1[j] = V2[j] + 2.1234;
}
```

```
__m128 num1, num2, num3;

num1 = _mm_set_ps (2.12, 2.12, 2.12, 2.12);

for (int i = 0; i < 8; i += 4) {
   num2 = _mm_load_ps (&V2[i]);
   num3 = _mm_add_ps (num1, num2);
   _mm_store_ps( &V1[i], num3);
}
```

```
for (j = 8; j < 10; j++) {
V1[j] = V2[j] + 2.1234;
}
```



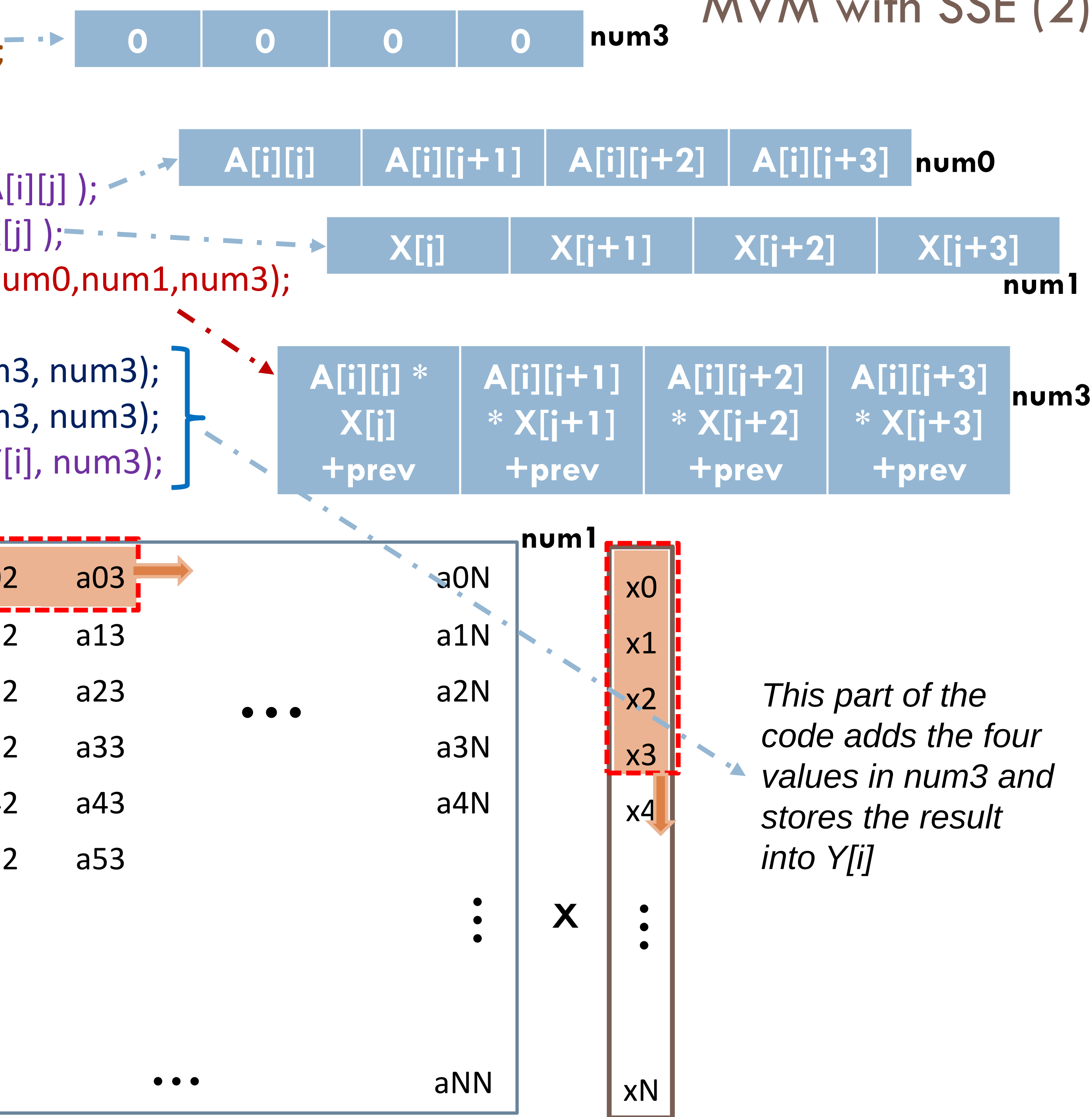
Case Study
MVM using SSE technology

```
float A[N][N];
float X[N], Y[N];
int i,j;

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    Y[i] += A[i][j] * X[j];
```

```
__declspec (align(16)) float  X[N], Y[N], A[N][N];
__m128 num0, num1, num2, num3, num4;

for (i=0; i<N;  i++){

num3= _mm_setzero_ps ();

 for (j=0; j<N; j+=4){
   num0 =_mm_load_ps ( &A[i][j] );
   num1=_mm_load_ps ( &X[j] );
   num3 =_mm_fmadd_ps (num0,num1,num3);
 }
num4 = _mm_hadd_ps (num3, num3);
num4 = _mm_hadd_ps (num4, num4);
_mm_store_ss( (float *) &Y[i] , num4);
}
```

MVM with SSE (2)

```
for (i=0; i!=N;  i++){
num3= _mm_setzero_ps();

 for (j=0; j!=N; j+=4){
  num0=_mm_load_ps( &A[i][j] );
  num1=_mm_load_ps( &X[j] );
  num3=_mm_fmadd_ps(num0,num1,num3);
 }
num3=_mm_hadd_ps(num3, num3);
num3=_mm_hadd_ps(num3, num3);
_mm_store_ss((float *) &Y[i], num3);
}
```

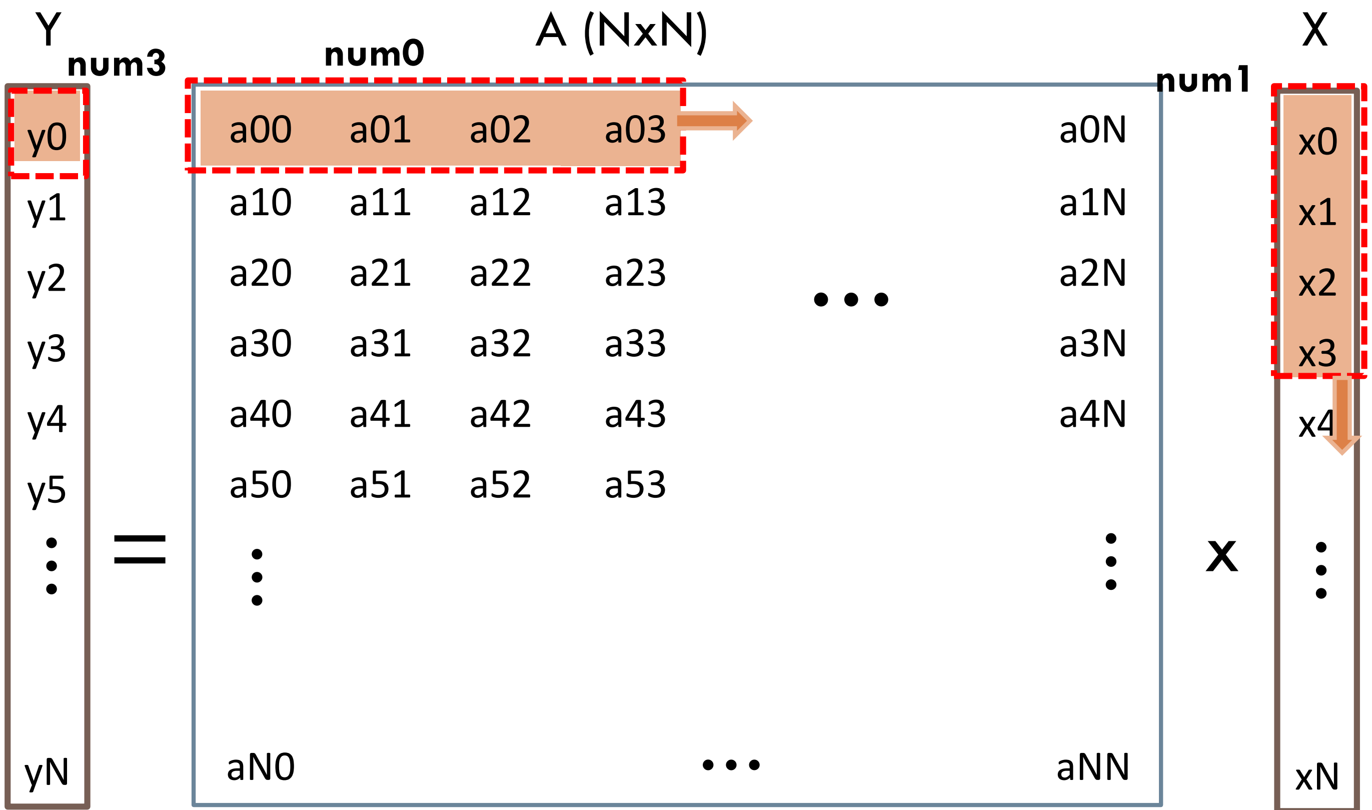


This part of the code adds the four values in num3 and stores the result into Y[i]

MVM with SSE (3)

```
....
}
num4=_mm_hadd_ps(num3, num3);
num4=_mm_hadd_ps(num4, num4);
_mm_store_ss((float *) &Y[i], num4);
}
```

- After j loop finishes its execution, num3 contains the output data of Y[0]
- let's say **num3=[ya, yb, yc, yd], we must compute this: Y[i]=ya+yb+yc+yd**
- after the **1st hadd -> num3=[ya+yb, yc+yd, ya+yb, yc+yd]**
- after the **2nd hadd -> num3=[ya+yb+yc+yd, ya+yb+yc+yd, ya+yb+yc+yd, ya+yb+yc+yd]**



Case Study
MVM using AVX technology

```
float A[N][N];
float X[N], Y[N];
int i,j;

for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  Y[i] += A[i][j] * X[j];
```

```
__m256  ymm2, num0, num1, num5;
__m128 xmm1, xmm2;

for (int i = 0; i < M; i++) {
num1 = _mm256_setzero_ps ();

for (int j = 0; j < M; j += 8) {
  num5 = _mm256_load_ps ( &X[j] );
  num0 = _mm256_load_ps ( &A[i][j] );
  num1 = _mm256_fmadd_ps (num0, num5, num1);
                         }
  ymm2 = _mm256_permute2f128_ps (num1, num1, 1);
  num1 = _mm256_add_ps (num1, ymm2);
  num1 = _mm256_hadd_ps (num1, num1);
  num1 = _mm256_hadd_ps (num1, num1);
  xmm2 = _mm256_extractf128_ps (num1, 0);
  _mm_store_ss ((float *) &Y[i] , xmm2);
            }
```

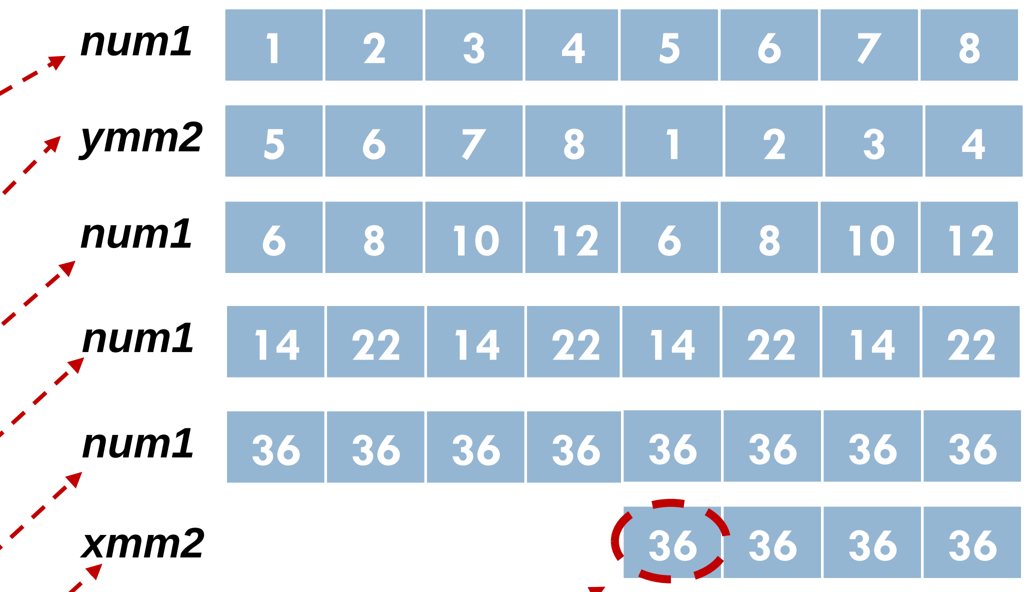
Case Study MVM
SSE vs AVX

```
__m128 num0, num1, num2, num3,
num4;

for (i=0; i<N;  i++){
num3= _mm_setzero_ps ();

 for (j=0; j<N; j+=4){
   num0 =_mm_load_ps ( &A[i][j] );
   num1=_mm_load_ps ( &X[j] );
   num3 =_mm_fmadd_ps (num0, num1,
num3);
 }
num4 = _mm_hadd_ps (num3, num3);
num4 = _mm_hadd_ps (num4, num4);
_mm_store_ss( (float *) &Y[i] , num4);
}
```

The different part

```
__m256  ymm2, num0, num1, num5;
__m128 xmm1, xmm2;

for (int i = 0; i < M; i++) {
num1 = _mm256_setzero_ps ();

for (int j = 0; j < M; j += 8) {
 num5 = _mm256_load_ps ( &X[j] );
 num0 = _mm256_load_ps (&A[i][j]);
 num1 = _mm256_fmadd_ps (num0, num5, num1);
                              }
   ymm2 = _mm256_permute2f128_ps (num1,
num1, 1);
   num1 = _mm256_add_ps (num1, ymm2);
   num1 = _mm256_hadd_ps (num1, num1);
   num1 = _mm256_hadd_ps (num1, num1);
   xmm2 = _mm256_extractf128_ps (num1, 0);
  _mm_store_ss ((float *) &Y[i] , xmm2);
              }
```

Further explaining the AVX code



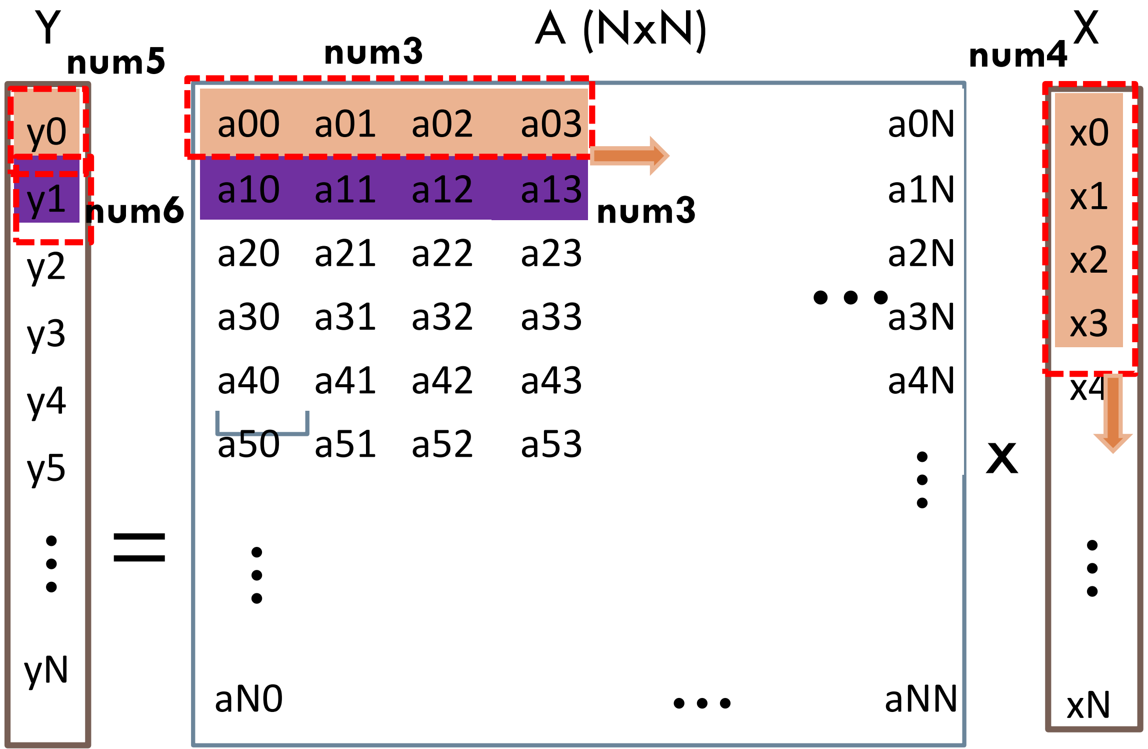
num1	1	2	3	4	5	6	7	8
ymm2	5	6	7	8	1	2	3	4
num1	6	8	10	12	6	8	10	12
num1	14	22	14	22	14	22	14	22
num1	36	36	36	36	36	36	36	36
xmm2					36	36	36	36

```
ymm2 = _mm256_permute2f128_ps (num1, num1, 1);
num1 = _mm256_add_ps (num1, ymm2);
num1 = _mm256_hadd_ps (num1, num1);
num1 = _mm256_hadd_ps (num1, num1);
xmm2 = _mm256_extractf128_ps (num1, 0);
_mm_store_ss ((float *) &Y[i], xmm2);
```

MVM with SSE + register blocking

```
for (i=0; i!=N;  i+=2){
num5= _mm_setzero_ps();
num6= _mm_setzero_ps();
 for (j=0; j!=N; j+=4){
  num3=_mm_load_ps( &A[i][j] );
  num4=_mm_load_ps(X + j );
  num5=_mm_fmadd_ps(num3,num4,num5);
  num3=_mm_load_ps( &A[i+1][j] );
  num6=_mm_fmadd_ps(num3,num4,num6);
 }
num5=_mm_hadd_ps(num5, num5);
num5=_mm_hadd_ps(num5, num5);
_mm_store_ss((float *)Y+i, num5);
num6=_mm_hadd_ps(num6, num6);
num6=_mm_hadd_ps(num6, num6);
_mm_store_ss((float *)Y+i+1, num6);

}
```

- **Register blocking is applied exactly as in the scalar code case**

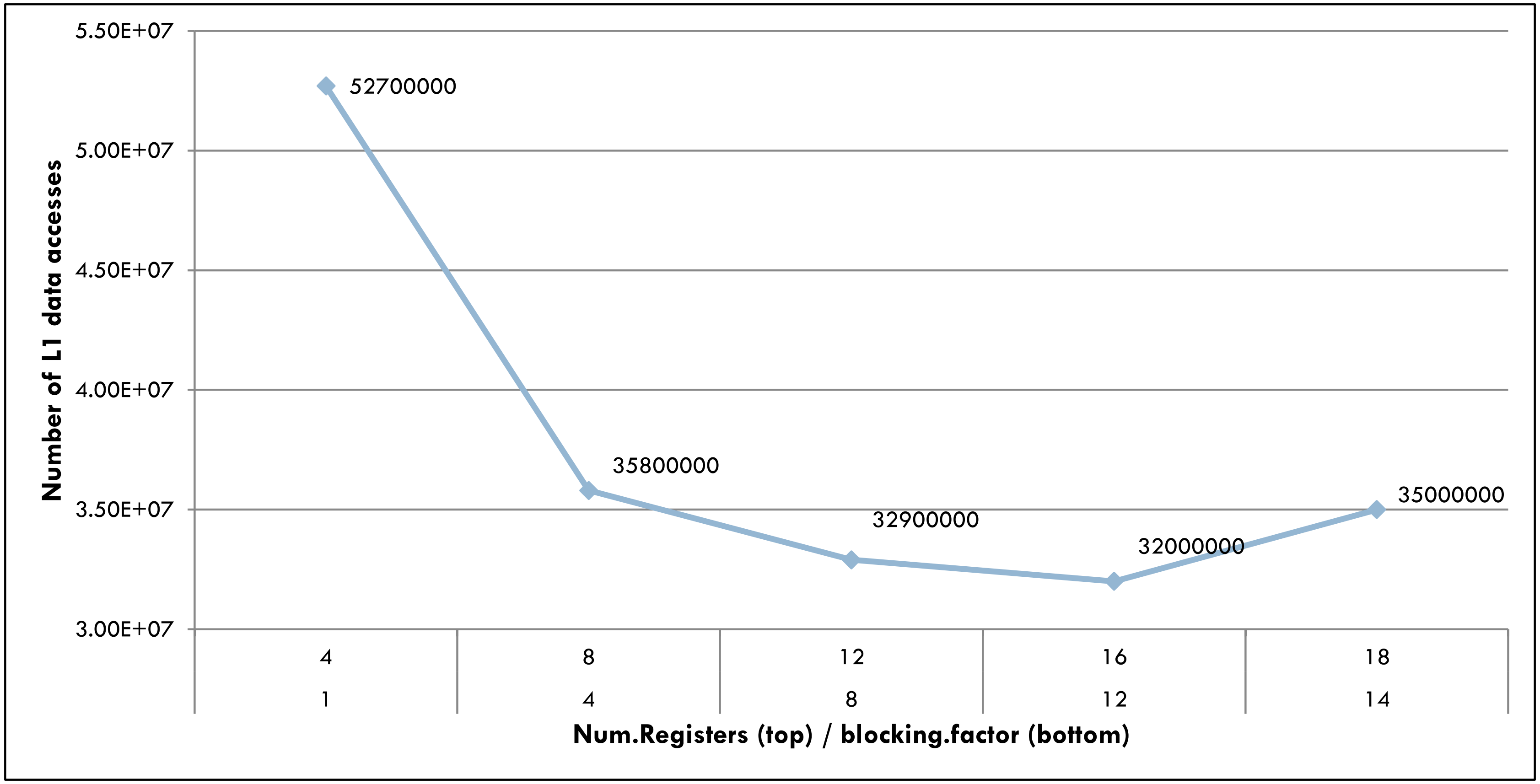


MVM with SSE + register blocking (1)
exercise from the lab this week

- Apply register blocking as in the previous slide with register blocking factor *[1, 2, 4, 8, 12, 14]*

- *Measure execution time for each code version*
  - *you will get a graph that looks like the one in the next slide*

MVM with SSE + register blocking (2)

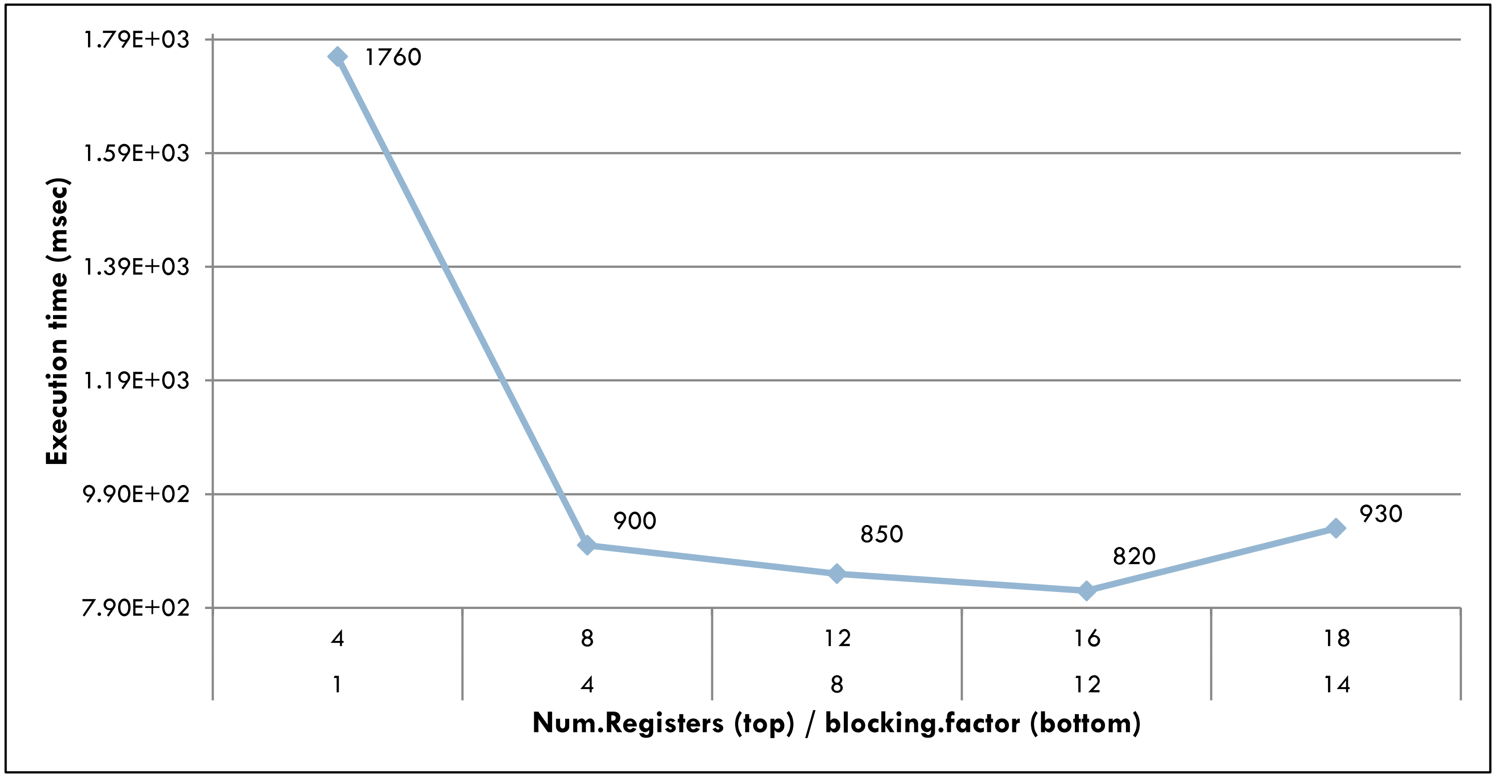


dL1 accesses:

$N+N^2+N^2$/unroll.factor.value

MVM with SSE + register blocking (3)

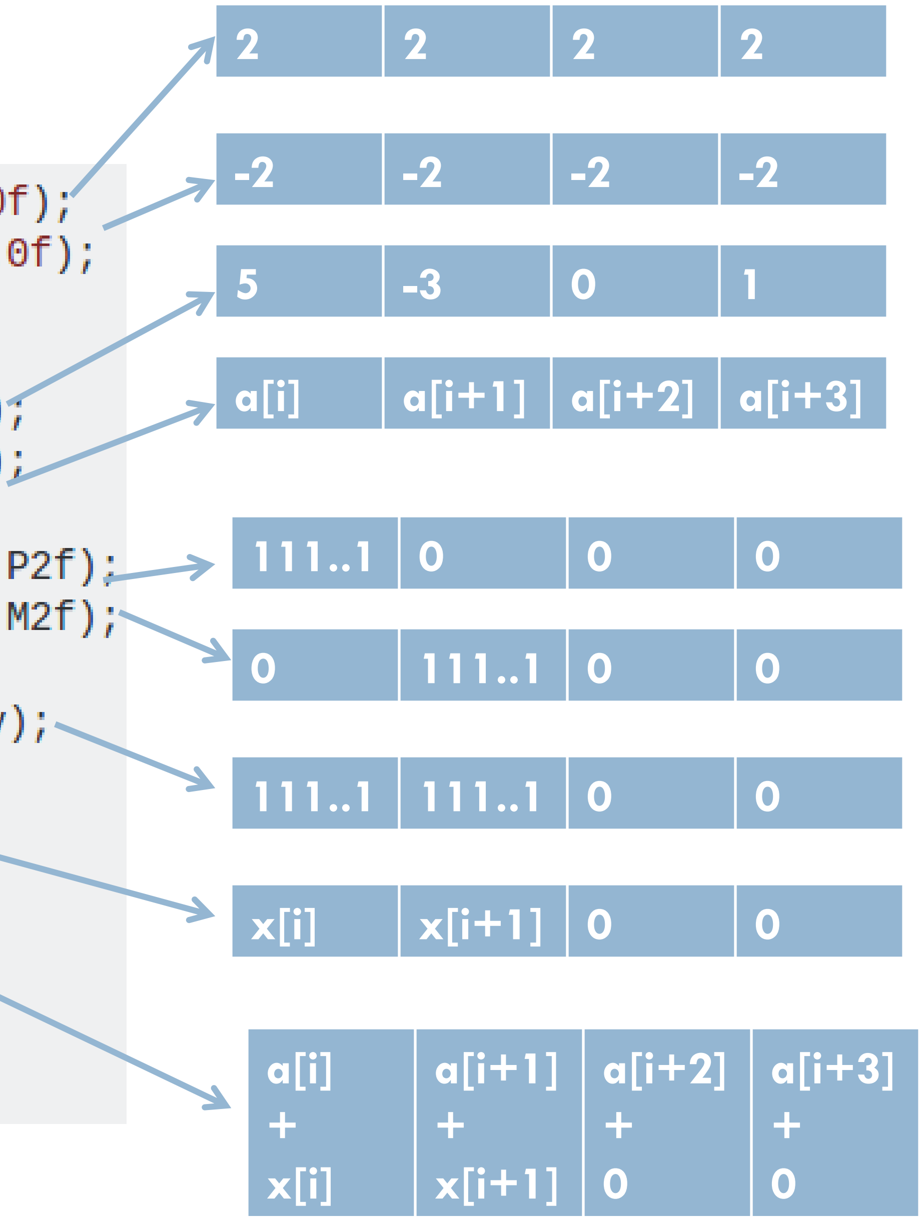
- Register blocking reduces the number of Load/Store (L/S) instructions and as a consequence the number of L1 data cache accesses
- Normally, performance highly depends on the L/S instructions, but there are other parameters that affect execution time too
  - This is **why you might get a different graph depending on the target processor and compiler**



What about if-conditions on SSE ?

```
for (i=0; i < n; i++) {
   if ( x[i] > 2 || x[i] < -2 )
      a[i]+=x[i];  }
```

```
const __m128 P2f = _mm_set1_ps(2.0f);
const __m128 M2f = _mm_set1_ps(-2.0f);
for (int i = 0; i < n; i += 4)
{
    __m128 xv = _mm_load_ps(x + i);
    __m128 av = _mm_load_ps(a + i);

    __m128 c1v = _mm_cmpgt_ps(xv, P2f);
    __m128 c2v = _mm_cmplt_ps(xv, M2f);

    __m128 cv = _mm_or_ps(c1v, c2v);

    xv = _mm_and_ps(xv, cv);

    av = _mm_add_ps(av, xv);

    _mm_store_ps(a + i, av);
}
```



Further Reading

- *Virtual Workshop (Cornell University), available at https://cvw.cac.cornell.edu/vector/overview_simd*
- *Tutorial from Virginia University, available at https://www.cs.virginia.edu/~cr4bd/3330/F2018/simdref.html*