CS420 – Lecture 6

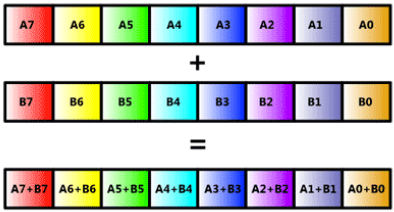
Raghavendra Kanakagiri
Slides: Marc Snir

Spring 2023

Parallelism

Vector operations

- *Single Instruction, Multiple Data* (SIMD): one instruction operates simultaneously on multiple entries of a vector.
- Uses *vector registers* and vector arithmetic units



Why vector operations?

- Reduce instruction decoding overhead
- Hardware does not need to worry about dependences between operations
  - It is up to the programmer and the compiler to create groups of independent adds
- Runs twice as fast if single precision is used

using single precision (with vector instructions) means twice as many flops – as well as better use of memory capacity and bandwidth and better cache hit ratio Should be done whenever higher precision is not required.

Vector instructions

Instruction Set	Name	Functionality	Year
MMX	Multimedia Extensions	64 bit MMX for packed integers	1997
SSE	Streaming SIMD Extensions	128 bit XMM for floating point.	1999
SSE2	Steaming SIMD Extensions 2	XMM supports doubles and integers	2001
SSE3	Streaming SIMD Extensions 3	Horizontal operations added	2004
SSSE3	Supplemental SSE3	Horizontal and data movement	2006
SSE4.1	Streaming SIMD Extensions 4.1	Extra functionality	2007
SSE4.2	Streaming SIMD Extensions 4.2	Vector string instructions	2008
AVX	Advanced Vector Extensions	256 bit YMM for floating point.	2011
FMA	Fused Multiply Add	Fused multiply add instructions	2011
AVX2	Advanced Vector Extensions 2	YMM supports packed integers	2013
AVX512	Advanced Vector Extensions 512	512 bit ZMM registers	2016

Table 1. Available x86 vector instruction sets

Vector instructions

Register	long double	double	float	long	int	short	char
64 bit MM	–	–	–	1	2	4	8
128 bit XMM	–	2	4	2	4	8	16
256 bit YMM	–	4	8	4	8	16	32
512 bit ZMM	–	8	16	8	16	32	64

Table 2. Vector lengths for different vector registers and data types

https://info.ornl.gov/sites/publications/files/Pub69214.pdf

How does one use vector instructions?

- Trust the compiler to *vectorize* loops (but verify [-qopt-report=5])
  - works well for simple loops
- Assembly language

How does one use vector instructions?

- OpenMP SIMD

```
#pragma omp simd
for (i=0;i<8;i++)
  c[i]=a[i]+b[i]
```

How does one use vector instructions?

- Use vector *intrinsics* (builtin functions recognized by the compiler); they map onto operations on vector registers.
  - Vector intrinsics are machine specific; we illustrate with Intel intrinsics

```
  void example(){
__m128 rA, rB, rC;
 for (int i=0; i<LEN; i+=4){
    rA = _mm_load_ps(&a[i]);
    rB = _mm_load_ps(&b[i]);
    rC = _mm_add_ps(rA,rB);
   _mm_store_ps(&C[i], rC);
}}
```

Programmer intervention

```
void test(float* A, float* B, float* C, float* D, float* E, int N)
{
  for (int i = 0; i <N; i++){
    A[i]=B[i]+C[i]+D[i]+E[i];
  }
}
```

Programmer intervention

```
float* a  = &b[1];
...
void func1(float *a,float *b, float *c)
{
   for (int i = 0; i < LEN; i++)
       a[i] = b[i] + c[i];
}
```

Programmer intervention

```
float* a  = &b[1];
...
void func1(float *a,float *b, float *c)
{
   for (int i = 0; i < LEN; i++)
       a[i] = b[i] + c[i];
}
```

```
a[0]= b[0] + c[0]
a[1] = b[1] + c[1]
```

is actually

```
b[1]= b[0] + c[0]
b[2] = b[1] + c[1]
```

Programmer intervention

```
void test(float* A, float* B, float* C, float* D, float* E, int N)
{
  for (int i = 0; i <N; i++){
    A[i]=B[i]+C[i]+D[i]+E[i];
  }
}
```

icc -std=c99 -O3 -xCORE-AVX512 -qopt-zmm-usage=high -qopt-report=5 test.c main.c

```
LOOP BEGIN at test.c(3,4)
 remark #15344: loop was not vectorized: vector dependence prevents
 vectorization
 remark #15346: vector dependence: assumed FLOW dependence between A[i] (4:6)
 and B[i] (4:6)
 remark #15346: vector dependence: assumed ANTI dependence between B[i] (4:6)
 and A[i] (4:6)
LOOP END
```

Programmer intervention

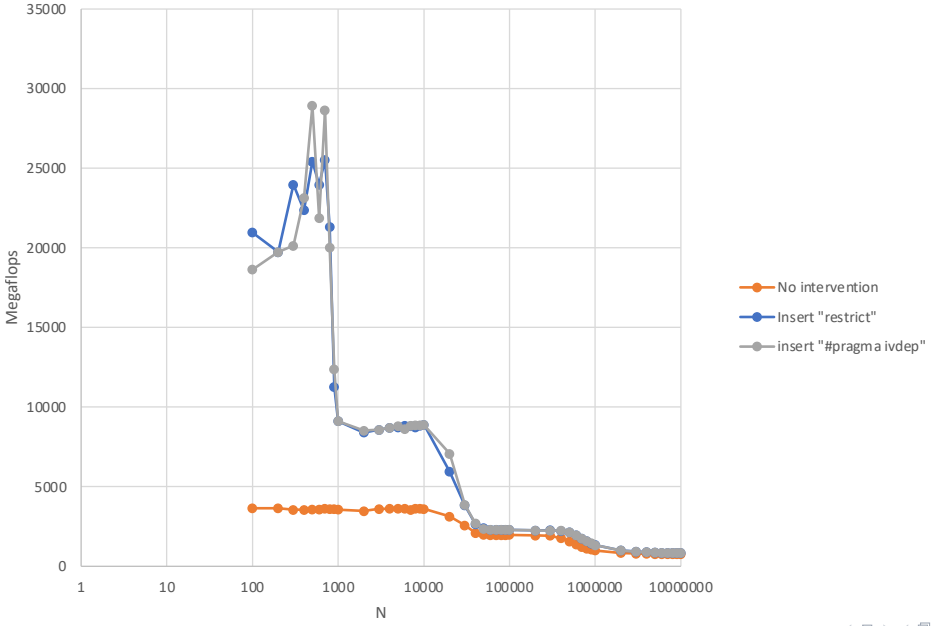
```
void test(float* __restrict__ A,
float* __restrict__ B,
float* __restrict__ C,
float* __restrict__ D,
float* __restrict__ E,
int N)
for (int i = 0; i <N; i++)
      A[i]=B[i]+C[i]+D[i]+E[i];
```

Programmer intervention

```
void test(float* A, float* B, float* C, float* D, float* E, int N)
#pragma ivdep
for (int i = 0; i < N; i++)
       A[i]=B[i]+C[i]+D[i]+E[i];
```

https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/pragmas/intel-specific-pragma-reference/ivdep.html

Programmer intervention



Data Alignment

- Vector loads/stores 128 consecutive bits to a vector register.
- Performance tends to be better when the addresses are 16-byte (128 bits) aligned for load/store
  - Intel platforms support aligned and unaligned load/store. Other systems may not.
  - Intel uses different instructions for aligned and unaligned accesses.

```
void test1(float *a,float *b,float *c)
{
for (int i=0;i<LEN;i++){
   a[i] = b[i] + c[i];
}
```

Data Alignment

- To know if a pointer is 16-byte aligned, the last digit of the pointer address in hex must be 0.
- Note that if b[0] is 16-byte aligned, and is a single precision array, then b[4] is also 16-byte aligned

```
__attribute__ ((aligned(16))) float  B[1024];
int main(){
  printf("%p, %p\n", &B[0], &B[4]);
}
Output: 0x7fff1e9d8580, 0x7fff1e9d8590
```

Data Alignment

- Manual 16-byte alignment can be achieved by forcing the base address to be a multiple of 16.

```
__attribute__ ((aligned(16))) float b[N];
float* a = (float*) memalign(16,N*sizeof(float));
```

- When the pointer is passed to a function, the compiler should be aware of where the 16-byte aligned address of the array starts.

```
void func1(float *a, float *b, float *c) {
    __assume_aligned(a, 16);
    __assume_aligned(b, 16);
    __assume_aligned(c, 16);
  for int (i=0; i<LEN; i++) {
      a[i] = b[i] + c[i];
  }
}
```

Data Alignment - AVX 512

```
double matA[4000][4000];
double matB[4000][4000];
double matC[4000][4000];

for (int j = 0; j < N; j++){
  for (int i = 0; i < M; i++){
    matA[j][i]+=matB[j][i]*matC[j][i];
  }
}
```

Data Alignment - AVX 512

```
   remark #25084: Preprocess Loopnests: Moving Out Store    [ mainru.c(41,30)]
   remark #15389: vectorization support: reference matA[i][j] has unaligned acce
[ mainru.c(42,17) ]
   remark #15389: vectorization support: reference matB[i][j] has unaligned acce
[ mainru.c(43,17) ]
   remark #15388: vectorization support: reference matC[i][j] has aligned access
[ mainru.c(44,17) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15305: vectorization support: vector length 8
   remark #15399: vectorization support: unroll factor set to 2
   remark #15309: vectorization support: normalized vectorization overhead 0.500
   remark #15300: LOOP WAS VECTORIZED
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15451: unmasked unaligned unit stride stores: 2
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 11
   remark #15477: vector cost: 2.000
   remark #15478: estimated potential speedup: 5.440
   remark #15487: type converts: 1
   remark #15488: --- end vector cost summary ---
   remark #25015: Estimate of max trip count of loop=250
   LOOP END
```

Data Alignment - AVX 512

```
__attribute__ ((aligned(64))) double matA[4000][4000];
__attribute__ ((aligned(64))) double matB[4000][4000];
__attribute__ ((aligned(64))) double matC[4000][4000];

for (int j = 0; j < N; j++){
  for (int i = 0; i < M; i++){
    matA[j][i]+=matB[j][i]*matC[j][i];
  }
}
```

Data Alignment - AVX 512

```
 LOOP BEGIN at mainr.c(56,5)
 remark #15388: vectorization support: reference matA[j][i] has aligned access
 remark #15388: vectorization support: reference matA[j][i] has aligned access
 remark #15388: vectorization support: reference matB[j][i] has aligned access
 remark #15388: vectorization support: reference matC[j][i] has aligned access
 remark #15305: vectorization support: vector length 8
 remark #15399: vectorization support: unroll factor set to 2
 remark #15300: LOOP WAS VECTORIZED
 remark #15448: unmasked aligned unit stride loads: 3
 remark #15449: unmasked aligned unit stride stores: 1
 remark #15475: --- begin vector cost summary ---
 remark #15476: scalar cost: 9
 remark #15477: vector cost: 0.870
 remark #15478: estimated potential speedup: 10.240
 remark #15488: --- end vector cost summary ---
 remark #25015: Estimate of max trip count of loop=252
LOOP END
```

Alignment in a struct

```
struct st {
  char A;
  int B[64];
   float C;
  int D[64];
};
int main(){
  st s1;
  printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D);
}
```

Output: 0x7fffe6765f00, 0x7fffe6765f04, 0x7fffe6766004, 0x7fffe6766008
The value of the addresses indicate that arrays B and D are not 16-bytes aligned.

Alignment in a struct

```
struct st{
  char A;
  int B[64] __attribute__ ((aligned(16)));
   float C;
  int D[64] __attribute__ ((aligned(16)));
};
int main(){
  st s1;
  printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D);
}
```

Output: 0x7fff1e9d8580, 0x7fff1e9d8590, 0x7fff1e9d8690, 0x7fff1e9d86a0
Arrays A and B are aligned to 16-bytes (notice the 0 in the 4 least significant bits of the address). Compiler automatically does padding.

Intel SSE Intrinsics **(Reference Slide)**

Datatypes:

- __m128 packed single precision (vector XMM register)
- __m128d packed double precision (vector XMM register)
- __m128i packed integer (vector XMM register)

```
  #include <xmmintrin.h>
int main ( ) {
  ...
  __m128 A, B, C;  /* three packed s.p. variables */
  ...
}
```

Intel SSE Intrinsics **(Reference Slide)**

Intrinsics operate on these types and have the format:

```
_mm_instruction_suffix(    )
```

Suffix can take many forms. Among them:

- ss scalar single precision
- ps packed (vector) singe precision
- sd scalar double precision
- pd packed double precision
- si scalar integer (8, 16, 32, 64, 128 bits)
- su scalar unsigned integer (8, 16, 32, 64, 128 bits)

Intel SSE Intrinsics: Example

```
  #define n 1024
  int main() {
    float a[n], b[n], c[n];
    for (i = 0; i < n; i+=4) {
    c[i:i+3]=a[i:i+3]+b[i:i+3];
   }
  }
```

```
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float a[n], b[n], c[n];
int main() {
__m128 rA, rB, rC;
for (i = 0; i < n; i+=4) {
  rA = _mm_load_ps(&a[i]);
  rB = _mm_load_ps(&b[i]);
  rC= _mm_mul_ps(rA,rB);
  _mm_store_ps(&c[i], rC);
}}
```

Vector operations **(Reference Slide)**

vector op vector apply operation element-wise

masked vector-vector apply operation element-wise where mask is 1

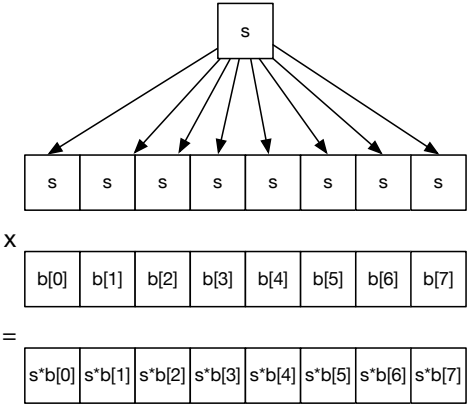
```
  register double a[8],b[8],c[8],d[8];
  register boolean mask[8];
  ...
for(i=0;i<8; i++)
   a[i]=mask[i]?b[i]+c[i]:d[i];

  ...
  __m512d a,b,c,d; __mmask8 mask;
  ...
  a=_mm512_mask_add(d,mask,b,c)
```

- masked vector operations take (at least) as much time as regular vector operations!

(Reference Slide)

vector op scalar done as broadcast, followed by vector-vector operation

```
register double a[4],s
  for(i=0;i<4; i++)
    a[i]=s*b[i];

__m512d ss,a,b; double s;
...
ss =_mm512_broadcastd_pd(s);
a=_mm512_mul_pd(ss,b)
```



Reduction **(Reference Slide)**

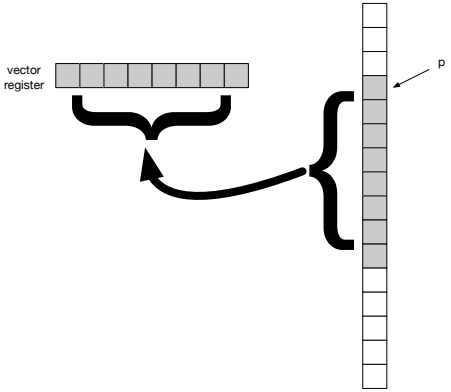
sum elements of a vector

```
register double a[8],s;
 for(i=0;i<8; i++)
   s=s+a[i];

...
__m512d a; double s;
s=_mmreduce_add_pd(a);
```

load/store **(Reference Slide)**

load/store from/to consecutive memory locations

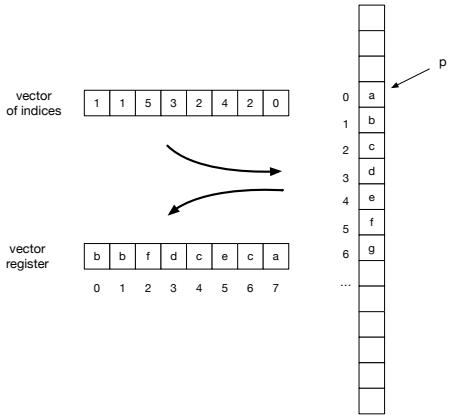
```
register a[8]; double *p;
...
for(i=0; i<8; i++)
   a[i] = p++


...
__m512d a; double *p
a = _mm512_load_pd(p)
```



gather-scatter **(Reference Slide)**

- gather nonconsecutive elements in memory to a vector register
- scatter elements in a vector register to nonconsecutive memory location

```
register double a[], b[8]; long p[8];
for(i=0;i<8; i++)
  b[i]=a[p[i]];

...
__mmd b; __m512i p; double a[];
b=_mm512_i64gather_pd (_p, a, 8);
```



- Much less efficient than load/store.
- May be required for irregular computations