Intrinsics for SIMD
Part 2

Satish Puri

Evaluate Polynomial

$3x^9 + x^7 + 12x^6 + 8x^5 + 7x^4 + x^2 - 9x - 6$

Array Index	9	8	7	6	5	4	2	1	0
coefficient	3	0	1	12	8	7	1	-9	-6
exponent	512	256	128	64	32	16	4	2	1
Term									

x = 2

```
double total = 0;
for(int i = 0; i< TERMS; i++)
{
    Term t = coefficient[i] * exponent[i]
    Add  t  to  total
}
```

Using intrinsics for Polynomial Evaluation

- double coeff[MAX] = {1,2,4, 8, 5, …, 88};
- double expon[MAX] = {512, 256, 128, 64, …, 4, 2,1 };
- double term[MAX]
- For(int t = 0; t< MAX; t++)

```
{
        term[t] = coeff[t] * expon[t];
}
```

Array program

```
void vecMulAdd(int n, double *A, double *B, double *C)
{
  int i;
  for(i = 0; i<n; i++)
  {
    C[i] = C[i] + A[i] * B[i];
  }
}
```

A	1	2	3	4	5	6	7	8
B	0	1	2	3	6	7	8	9
C	0	0	0	0	0	0	0	0
	0	2	6	12	30	42	56	72

Translate the code using SIMD Intrinsics

```
void vecMulAdd(int n, double *A, double *B, double *C)
{
  int i;
  for(i = 0; i<n; i++)
  {
    C[i] = C[i] + A[i] * B[i];
  }
}
```

A	1	2	3	4	5	6	7	8
B	0	1	2	3	6	7	8	9
C	0	0	0	0	0	0	0	0
	0	2	6	12	30	42	56	72

Memory alignment requirements

For doubles, 32 bytes aligned

memory address starts at modulo 32

_mm_malloc(amount of memory, alignment)

ensures that data is aligned to a 32 byte boundary.

- An instruction that attempts to read or write unaligned data can cause a segmentation fault, indicating an invalid memory reference.

- Poor performance if not aligned.

Alignment question 1

Let us assume that **arr** array is stored in a block of memory with starting location $(3200)_{10}$

```
double *arr = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
```

Assuming double occupies 8 bytes of memory space, will this line of code work?

```
__m256d first = _mm256_load_pd(arr + 1);
```

a) True
b) False

Aligning static arrays

double c[256]; May or may not be aligned

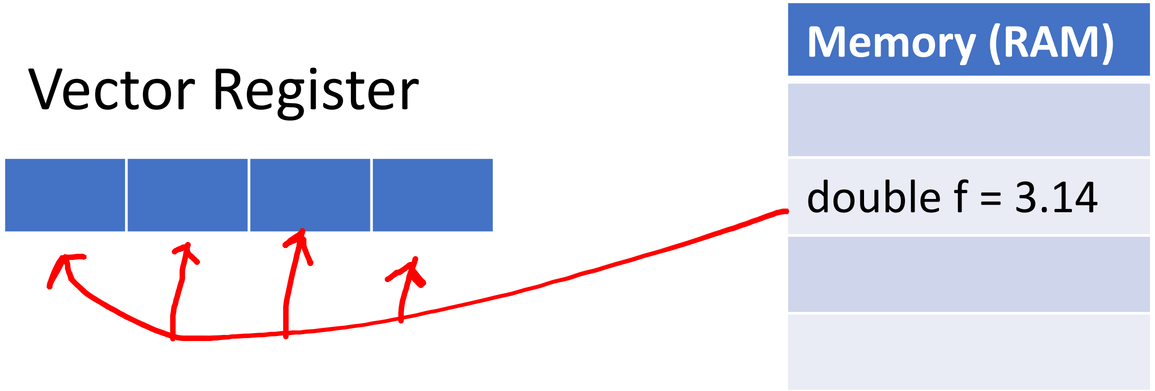
Forcing alignment

```
double c[256]   __attribute__((aligned(32)));
```

array c's starting address aligned by 32 bytes

Broadcast

- Loads and broadcast a value to fill a vector register



- How to copy the same value in all the elements of register?

- _mm256_broadcast_sd(address of double variable)
  returns the vector

```
double *arr = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
int i = 2;
__m256d c0 = _mm256_broadcast_sd( arr + i);
```

Same as:

```
__m256d c0 = _mm256_broadcast_sd( &arr[i]);
```

What does c0 contain?

a) 1.0
b) 1.0, 1.0, 1.0, 1.0
c) 3.0
d) 3.0, 3.0, 3.0, 3.0

Initialize a vector type

```
__m256d c0 = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);
```

Similar to initialization

c0 now contains {0, 0, 0, 0}

c0

Scalar and Vector Instruction

_mm256_broadcast_sd

_mm256_load_pd

X: **s** stands for **Scalar** 1 data

p stands for **Packed** multiple data

Alignment not an issue for scalar instruction

Alignment question 2

Let us assume that arr array is stored in a block of memory with starting location $(3200)_{10}$

double *arr = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};

Assuming double occupies 8 bytes of memory space, will this line of code work?

```
__m256d first = _mm256_broadcast_sd(arr + 1);
```

a) True
b) False

Matrix Multiplication

The general pattern for matrix multiplication is as follows.

$$
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\
a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\
a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\
\vdots & \vdots & \vdots & & \vdots \\
a_{i1} & a_{i2} & a_{i3} & \cdots & a_{in} \\
\vdots & \vdots & \vdots & & \vdots \\
a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
b_{11} & b_{12} & \cdots & b_{1j} & \cdots & b_{1p} \\
b_{21} & b_{22} & \cdots & b_{2j} & \cdots & b_{2p} \\
b_{31} & b_{32} & \cdots & b_{3j} & \cdots & b_{3p} \\
\vdots & \vdots & & \vdots & & \vdots \\
b_{n1} & b_{n2} & \cdots & b_{nj} & \cdots & b_{np}
\end{bmatrix}
=
\begin{bmatrix}
c_{11} & c_{12} & \cdots & c_{1j} & \cdots & c_{1p} \\
c_{21} & c_{22} & \cdots & c_{2j} & \cdots & c_{2p} \\
\vdots & \vdots & & \vdots & & \vdots \\
c_{i1} & c_{i2} & \cdots & c_{ij} & \cdots & c_{ip} \\
\vdots & \vdots & & \vdots & & \vdots \\
c_{m1} & c_{m2} & \cdots & c_{mj} & \cdots & c_{mp}
\end{bmatrix}
$$

$$a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + \cdots + a_{in}b_{nj} = c_{ij}$$

Storing matrix in memory

Assumption: Matrices are stored in column-major order

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Storage is linear – put one column after another column.

1 5 9 13 2 6 10 14 3 7 11 15 4 8 12 16

Matrix Matrix Multiply

- Unoptimized code:

```
void dgemm (int n, double* A, double* B, double* C)
{
 for (int i = 0; i < n; ++i)
   for (int j = 0; j < n; ++j)
   {
    double cij = C[i+j*n];         /* cij = C[i][j] */
    for(int k = 0; k < n; k++ )
     cij += A[i+k*n] * B[k+j*n];     /* cij += A[i][k]*B[k][j] */
    C[i+j*n] = cij;       /* C[i][j] = cij */
   }
}
```

X86-64 Registers

- Regular integer registers

  Examples

  %r11 (64 bits)

  %rcx (64 bits)

  %rsi (64 bits)

  %eax (32 bits)

  %ebx (32 bits)

Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

How to read a vector instruction?

Format nameXY

X: **s** stands for **Scalar**

p stands for **Packed**

Y: Y is for data type. We will use only double so Y = d

vadd**s**d : **scalar** double

vadd**p**d : **packed** double

vmul**s**d : **scalar** double

vmul**p**d : **packed** double

Matrix Multiply

- x86 assembly code:

```
1. vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
2. mov %rsi,%rcx         # register %rcx = %rsi
3. xor %eax,%eax         # register %eax = 0
4. vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
5. add %r9,%rcx          # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1      # Multiply %xmm1, element of A
7. add $0x1,%rax         # register %rax = %rax + 1
8. cmp %eax,%edi         # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0          # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>   # jump if %eax > %edi
11. add $0x1,%r11d       # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10) # Store %xmm0 into C element
```

Matrix Multiply

- **Optimized C code:**

```c
#include <immintrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0,      /* c0 += A[i][k]*B[k][j] */
                _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
                _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0);      /* C[i][j] = c0 */
  }
}
```

Matrix Multiply

- **Optimized x86 assembly code:**

```
1. vmovapd (%r11),%ymm0           # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx                  # register %rcx = %rbx
3. xor %eax,%eax                  # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax                  # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1      # Parallel mul %ymm1,4 A elements
7. add %r9,%rcx                   # register %rcx = %rcx + %r9
8. cmp %r10,%rax                  # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0       # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>           # jump if not %r10 != %rax
11. add $0x1,%esi                 # register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)          # Store %ymm0 into 4 C elements
```

Comparison

Unoptimized	Optimized

Scalar double (sd) instructions
`vmovsd (%r10),%xmm0`
`# Load 1 element of C into %xmm0`
 	
`vmulsd`
 	
xmm register = 128 bits wide
	Parallel double (pd) instructions
	`vmovapd (%r11),%ymm0`
	`# Load 4 elements of C into %ymm0`
	
	`vmulpd`
	
	ymm register = 256 bits wide
	
	3 to 4 times faster by using Intrinsics

Execution time measurement

```
#include <sys/time.h>

struct timeval  tv1, tv2;

gettimeofday(&tv1, NULL);

  dgemm(dimension, A, B, C);

gettimeofday(&tv2, NULL);

double time_spent = (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 +
                    (double) (tv2.tv_sec - tv1.tv_sec);
```

HW: GFLOPS calculation

- Use matmul_intrinsics.c for testing your code
  - It has the logic and code for timing and gflops.
  - testing is same as matrix multiplication, you can reuse the code for array creation, initialization.

```
double time_spent = (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 +
                             (double) (tv2.tv_sec - tv1.tv_sec);

printf("Intrinsics: Time %f \n", time_spent);

double numOps = 2.0 * SIZE * SIZE * SIZE;

double gflops = 1.0e-9 * numOps / time_spent;

printf(" Intrinsics: numOps = %.2f GFLOPS %f \n", numOps, gflops);
```

- Code-reuse from matmul intrinsics code

```
int matmul_intrinsics()
{

  // int NUM_ELEMENTS = 1024*1024;
  //    int dimension = 1024;

  int NUM_ELEMENTS = SIZE*SIZE;
  int dimension = SIZE;
  printf("dimension = %d & NUM_ELEMENTS %d, sizeof(double)=%lu \n ", dir

  size_t N_pd = (NUM_ELEMENTS*8)/sizeof(double);

  double *data_A = (double*)_mm_malloc(N_pd*sizeof(double), 32);
  double *data_B = (double*)_mm_malloc(N_pd*sizeof(double), 32);
  double *data_C = (double*)_mm_malloc(N_pd*sizeof(double), 32);

  if(data_A == NULL || data_B == NULL || data_C == NULL)
  {
    printf("Error \n");
    return 1;
  }

  initialize(data_A, NUM_ELEMENTS, 1.0);
  //initialize(data_B, NUM_ELEMENTS, 2.0);
  initializeMM(data_B, NUM_ELEMENTS);
  initialize(data_C, NUM_ELEMENTS, 0.0);

  struct timeval  tv1, tv2;

  gettimeofday(&tv1, NULL);

  //dgemmIntrin(dimension, data_A, data_B, data_C);

   vecAddIntrin(dimension, data_A, data_B, data_C);

  gettimeofday(&tv2, NULL);

  double time_spent = (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 +
                                  (double) (tv2.tv_sec - tv1.tv_sec);

  printf("Intrinsics: Time %f \n", time_spent);
```

Compiling on Pascal

- Use Pascal server (Intel Processor required)

- Header file required
  #include<immintrin.h>

- gcc -O3 -mavx arrayIntrinsics.c
  OR
  gcc -O3 -o prog -mavx matmul_intrinsic.c

- ./a.out

Vectorization by compilers

- Compiler options/flags

  gcc –O2

  gcc –O3

- Vectorization is parallelization.
- Look at assembly for vector instructions
- Compiler vectorization report
- Compiler may do sub-optimal vectorization