Lab 6 of SP19-BL-ENGR-E599-30563

The high performance of DAAL kernel in Harp-DAAL comes from the leveraging of hardware resources like cores and vector registers. Except for the thread-level optimization such as thread scheduling, DAAL kernels explore more on the register resources via vectorization. In general, daal kernels achieve vectorization in two ways. 1) Invoke pre-compiled linear algebra libraries such as the Intel MKL, 2) Rely on compiler to automatically vectorize the loop structure in codes. We will cover the fundamental knowledge of vectorization in high performance codes as well as hands-on of vectorizing a matrix-vector multiplication example by using Intel's compiler.

Goal

- Basic knowledge of the vectorization
- Hands on of automatic vectorization by compiler

Deliverables

Submit a table (xlsx or plain csv txt file) that records the results of experiments as follows

- Run `matVecMul` with 100 iterations but with different matrix sizes, compare performance of `vectest-novec` with `vectest-avx`
  - Row=100, Col=100
  - Row=1000, Col=1000
  - Row=10000, Col=10000
- Run `matVecMul` with matrix sizes (10000×10000) but with increasing iterations, compare performance of `vectest-novec` with `vectest-avx`
  - Iterations = 10
  - Iterations = 100
  - Iterations = 1000
- Run the *pitfall* functions with default matrix size and iterations (10000, 10000, 100) and compare performance of `vectest-novec` with `vectest-avx`
  - `matVecMulNonCountable`
  - `matVecMulFCall`
  - `matVecMulIndirect`
  - `matVecMulBranch`

Evaluation

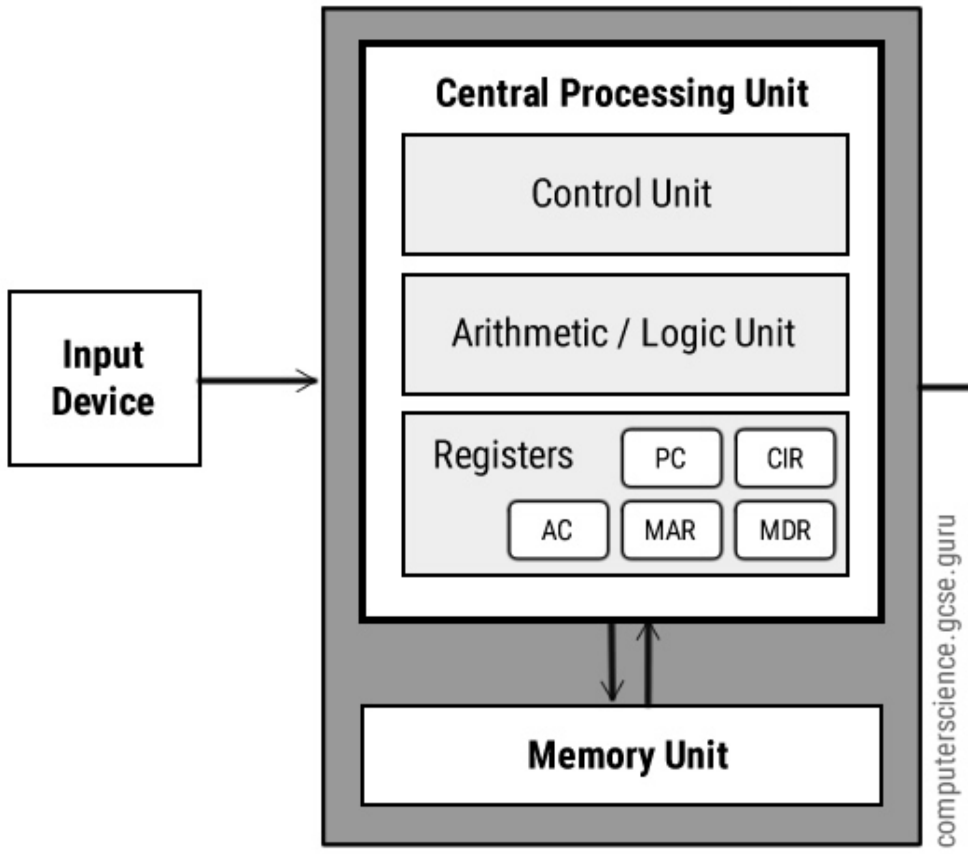
Lab participation: credit for 1 point based upon a successful completion of the lab tasks

Prerequisite Knowledge of Vectorization

- What is the vector register ?
- What is the SIMD ?

Vector register

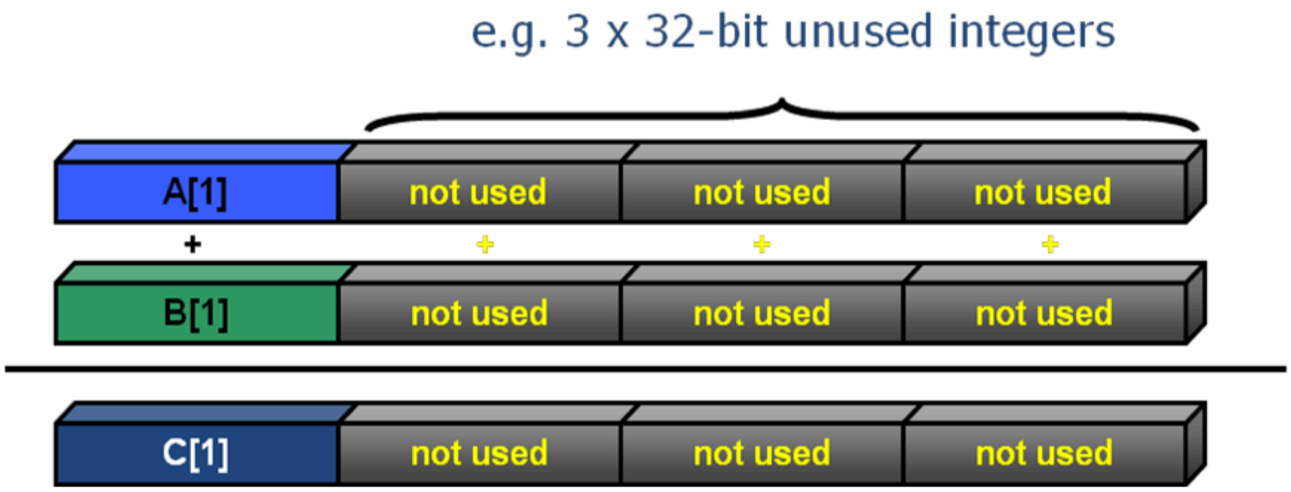
Registers in classic Von Neumann Architecture are electronic units that could hold a number of bits.



Register in CPU Architecture

As registers are packed beside the control unit and arithmetic logic unit on a CPU chip, CPU has a much lower latency of accessing data from register than from cache or main memory. Therefore, the registers are substantially leveraged in the high performance computing area.

A register in CPU usually has a wide of more than 64 bits. For instance, a register has 128 bits, it is able to hold 4 integer number. However, the scalar operation can only work on one integer at a time, which causes a waste of the other three integer slots in a 128-bit register.

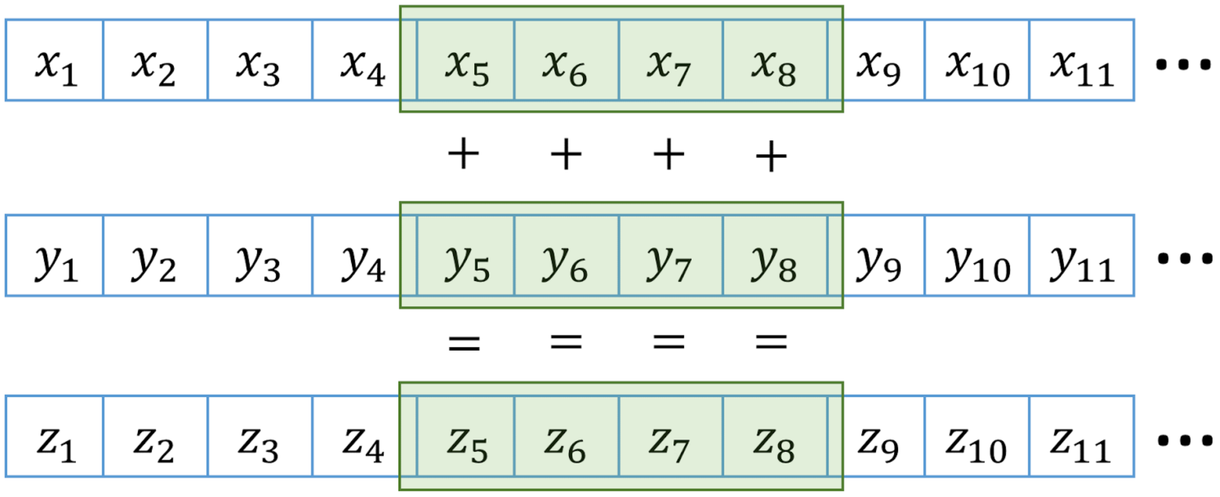


If the operations support data manipulation on multiple numbers concurrently, we could fully utilize all of the register wide and such operations are viewed as vector processing.

Single Instruction Multiple Data (SIMD) and AVX extension

Vector processing is also called Single Instruction Multiple Data (SIMD) in many scenarios, which is a way to explore the instruction-level parallelism (data parallelism) by executing the same operations on different pieces of data.

A simple example is the addition of two vectors


x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} ...
+ + + +
y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} ...
= = = =
z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8 z_9 z_{10} z_{11} ...

Suppose we have vector x and y with a length of 11 single precision numbers, and the register has a length of 128 bits.

- A non-vectorized code requires 22 operations to load x and y data from memory into register, 11 addition operations and another 11 write operations to write data from register back to memory.

```
for(int i=0;i<11;++i)
{
    z[i] = x[i] + y[i];
}
```

Although our register has a wide of 128 bits, the non-vectorized code only uses 16 bits in each loop iteration.

- A vectorized code just requires 6 operations of loading x and y data, 3 operations to apply addition and another 3 operations to write back data
    - The first loop iteration loads x[1-4], y[1-4], add them and write results to z[1-4]
    - The second loop iteration loads x[5-8], y[5-8], add them and write results to z[5-8]
    - The third loop iteration loads x[9-11], y[9-11], add them and write results to z[9-11]

Therefore the vectorization saves 16 load operations, 8 addition operations, and 8 write operations compared to the non-vectorized codes.

- Save the computation time
- Save the memory access time (reduce the latency and leverage the bandwidth)
- May not be efficient if the data access pattern is not regular (e.g., finding a key in a linked list)
- Programmers shall know about the hardware features (e.g., vector length)

Intel AVX

Advanced Vector Extensions (AVX, also known as Sandy Bridge New Extensions) are extensions to the x86 instruction set architecture for microprocessors from Intel and AMD proposed by Intel in March 2008 and first supported by Intel.

AVX starts from supporting 256 bits (compared to the legacy SSE that only supports 128 bits) in the AVX2 extension, and now it supports 512 bits register in the AVX-512 for the latest Xeon and Xeon Phi processors.

Compiler-based Automatic Vectorization

There are two ways of producing vectorized codes running on registers.

- Explicit vectorization by writing hand-coded intrinsic or assembly functions
    - E.g., Intel AVX intrinsics
- Automatic vectorization of loops by compiler
    - E.g., Intel icc compiler, gcc

While the explicit vectorization gives more fine-grained control on the generated codes, it lacks cross-platform portability in terms of hardware, OS, and compiler. Therefore, we recommend to use the compiler-based vectorization as a priority. In the rest of the session, we will focus on the automatic vectorization on Intel's hardware platform.

Matrix Vector Multiplication

A code example of matrix-vector multiplication is located at `/share/jproject/lc37/simd-tutorial` Please copy it to your home director

```
cp -r /share/jproject/lc37/simd-tutorial ~/
```

Make sure that you have your icpc compiler added to the PATH

```
source /opt/intel/compilers_and_libraries_2019/linux/bin/compilervars.sh intel64
```

Check the `icpc` compiler

```
which icpc
```

Compilation

We use `make` to compile the example codes

```
## compile

CXXFLAGS=-Wall -qopt-report=5
CC=icpc

all:
    ${CC} ${CXXFLAGS} -no-vec -o vectest-novec main.cpp
    ${CC} ${CXXFLAGS} -xCORE-AVX2 -o vectest-avx main.cpp

novec:
    ${CC} ${CXXFLAGS} -no-vec -o vectest-novec main.cpp

vecavx:
    ${CC} ${CXXFLAGS} -xCORE-AVX2 -o vectest-avx main.cpp

clean:
    rm vectest-*
```

, where the `CXXFLAGS` is the compilation flags, which contains the `-qopt-report=5` option to enable the optimization report returned by the `icpc` compiler. We shall compile two versions of binaries.

- *vectest-novec* is the binary without any vectorization optimization (with `-no-vec`)
- *vectest-avx* is the binary with AVX vectorization (with `-xCORE-AVX2`)

To compile both of the two binaries

```
make all
```

Code Walk Through

In the `main.cpp` file, you will find a simple implementation of the iterative dense matrix vector multiplication

$$C = \sum_{i=0}^{i=p} AB$$

, where A is a $m \times n$ dense matrix, B is a n-dimensional dense vector, and C is the summation of p iterations of AB. The pieces of codes are shown as follows

```
void matVecMul(float** m, float* y, float* z, int row_len, int col_len, int iters)
{
    __assume_aligned(y, 64);
    __assume_aligned(z, 64);

    for(int i=0;i<iters;i++)
        for(int j=0;j<row_len;j++)
        {
            __assume_aligned(m[j], 64);
            for(int k=0;k<col_len;k++)
                z[j] += m[j][k]*y[k];
        }

}
```

There are three for loops. The innermost loop computes the multiplication of

$$C[i] = \sum_{k=0}^{k=n} A[i,k]B[k]$$

The middle loop iterates over all rows of $A[i,:]$, and the outermost loop is the number of iteration p. There are also the alignment of data block to 64 bits, which improves the latency of fetching data from the memory to vector registers.

```
__assume_aligned(y, 64);
__assume_aligned(z, 64);
```

In the main function, we will create and initialize a matrix and a vector with specified dimensions.

```
int main(int argc, char** argv)
{

    int row_len = 10000;
    int col_len = 10000;
    int iters = 100;

    if (argc >1)
        row_len = atoi(argv[1]);

    if (argc >2)
        col_len = atoi(argv[2]);

    if (argc >3)
        iters = atoi(argv[3]);

    printf("Row length: %d\n", row_len);
    printf("Col length: %d\n", col_len);
    printf("Iterations: %d\n", iters);

    float** m = (float**) malloc(row_len*sizeof(float*));
    for(int i=0;i<row_len;i++)
    {
        m[i] = (float*) _mm_malloc(col_len*sizeof(float), 64);
        vecInit(m[i], col_len, 2);
    }

    float* y = (float*) _mm_malloc(col_len*sizeof(float), 64);
    vecInit(y, col_len, 3);
    float* z = (float*) _mm_malloc(row_len*sizeof(float), 64);
    vecInit(z, row_len, 0);

    ...
}
```

, where the matrix A is initialized with value 2 for all of its entries while the vector y is initialized with value 3. The default dimension is 10000×10000 with an iteration number of 100. You may also feed these values by using command line arguments like

```
./vectest-avx 10000 10000 100
```

Since the matrix and vector entries are stored in single precision floats (32 bits), make sure that the dimension of matrix will not exceed the capacity of your node memory resource during the experimentation.

The rest of the main function is to invoke the `matVecMul` function and record the elapsed time

```
double time = 0.0;
time = timer();
matVecMul(m, y, z, row_len, col_len, iters);
// matVecMulFCall(m, y, z, row_len, col_len, iters);
// matVecMulNonCountable(m, y, z, row_len, col_len, iters);
// matVecMulIndirect(m, y, z, row_len, col_len, iters);
// matVecMulBranch(m, y, z, row_len, col_len, iters);
printf("Time for mat vec mul: %f s\n", (timer() - time));
```

Along side the function `matVecMul`, we also have four variants of `matVecMul` to show four pitfall examples that compromise the performance of vectorization.

Non-Countable Loop Iterations

If the loop count is unknown in runtime, which may change during the execution of the loop, the performance of vectorization may be compromised.

```
void matVecMulNonCountable(float** m, float* y, float* z, int row_len, int col_len, int iters)
{
    __assume_aligned(y, 64);
    __assume_aligned(z, 64);

    int* loopCount = (int*)malloc(row_len*sizeof(int));
    for(int i=0;i<row_len;i++)
        loopCount[i] = col_len;

    for(int i=0;i<iters;i++)
        for(int j=0;j<row_len;j++)
        {
            __assume_aligned(m[j], 64);
            for(int k=0;k<loopCount[j];k++)
                z[j] += m[j][k]*y[k];
        }

    free(loopCount);
}
```

Function Call Within Loop

If the inner loop does call a non-inlined function, the compiler will not vectorize the codes

```
__attribute__((noinline)) float multiply(float a, float b)
{
    return a*b;
36  }
37
38  void matVecMulFCall(float** m, float* y, float* z, int row_len, int col_len, int iters)
39  {
40      __assume_aligned(y, 64);
41      __assume_aligned(z, 64);
42
43      for(int i=0;i<iters;i++)
44          for(int j=0;j<row_len;j++)
45          {
46              __assume_aligned(m[j], 64);
47              for(int k=0;k<col_len;k++)
48                  z[j] += multiply(m[j][k], y[k]);
49          }
50  }
```

The `icpc` compiler report

```
remark #15489: --- begin vector function matching report ---
remark #15490: Function call: multiply(float, float) with simdlen=4, actual parameter types: (vector,vector)   [ main.cpp(48,25) ]
remark #15545: SIMD annotation was not seen, consider adding 'declare simd' directives at function declaration
remark #15493: --- end vector function matching report ---
```

Indirect Memory Address

If the access to a memory location is indirect (indexed by another array), there will be additional overhead to the vectorization.

```
 90  void matVecMulIndirect(float** m, float* y, float* z, int row_len, int col_len, int iters)
 91  {
 92      __assume_aligned(y, 64);
 93      __assume_aligned(z, 64);
 94
 95      // an indirect memory access will prevent compilers from vectorizing codes
 96      int* index = (int*)malloc(col_len*sizeof(int));
 97      for(int i=0;i<col_len;i++)
 98          index[i] = i;
 99
100      for(int i=0;i<iters;i++)
101          for(int j=0;j<row_len;j++)
102          {
```

```
103                __assume_aligned(m[j], 64);
104                for(int k=0;k<col_len;k++)
105                    z[j] += m[j][index[k]]*y[index[k]];
106            }
107
108
109        free(index);
110    }
```

The `icpc` compiler report

```
remark #15415: vectorization support: irregularly indexed load was generated for the variable <*(*(m+j*8)+(*(index+k*4))*4)>, part of index is read from memory   [ main.cpp(105,25) ]
remark #15415: vectorization support: irregularly indexed load was generated for the variable <y[*(index+k*4)]>, part of index is read from memory   [ main.cpp(105,40) ]
```

Data-dependent Exit

If there is branch statement that triggers exit to a loop according to the data value, the compiler will not vectorize the codes.

```
72  void matVecMulBranch(float** m, float* y, float* z, int row_len, int col_len, int iters)
73  {
74      __assume_aligned(y, 64);
75      __assume_aligned(z, 64);
76
77      for(int i=0;i<iters;i++)
78          for(int j=0;j<row_len;j++)
79          {
80              __assume_aligned(m[j], 64);
81              for(int k=0;k<col_len;k++)
82              {
83                  if (y[k] > 1)
84                      z[j] += m[j][k]*y[k];
85              }
86          }
87
88  }
```

The `icpc` compiler report

```
remark #15344: loop was not vectorized: vector dependence prevents vectorization
remark #15346: vector dependence: assumed FLOW dependence between z[j] (84:21) and
z[j] (84:21)
remark #15346: vector dependence: assumed ANTI dependence between z[j] (84:21) and
z[j] (84:21)
```

Testing the codes

We use the `matVecMul` function and compile the codes, the compiler report shows

```
remark #15300: LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 1
remark #15450: unmasked unaligned unit stride loads: 1
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 11
remark #15477: vector cost: 1.000
remark #15478: estimated potential speedup: 7.360
remark #15488: --- end vector cost summary ---
```

By running `./vectest-novec` and `./vectest-avx`, we will have the screen output as

```
[lc37@j-006 simd-tutorial]$ ./vectest-novec
Row length: 10000
Col length: 10000
Iterations: 100
Time for mat vec mul: 9.778775 s
[lc37@j-006 simd-tutorial]$ ./vectest-avx
Row length: 10000
Col length: 10000
Iterations: 100
Time for mat vec mul: 1.129134 s
[lc37@j-006 simd-tutorial]$
```

It is shown that the vectorized codes run 9x faster than the non-vectorized codes