# Task 1 and Task 2

The SSE (Streaming SIMD Extensions) instruction set extension for the x86 architecture enables 128-bit wide SIMD operations. These operations enable the use of 4x32 bit/2x64 bit special purpose registers for performing the same operation on different data, in parallel. Therefore, using this approach should bring a theoretical improvement of 4 times in mathematical computations, when using single-precision floating-point or 2 times improvement when using double precision.

From our measurements (presented in Figure 1), the speed-up increases towards 3.4 when we reach the limit of RAM use (over 4 GB).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Matrix x vector - original implementation | | | | Implementation with arbitrary N | | | |
| N (matrix / vector size) | Sequential | SSE implementation (seconds) | Speed-up | N (matrix / vector size) | Sequential (seconds) | SSE implementation (seconds) | Speed-up |
| 500 | 0.001741 | 0.0007 | 2.487143 | 499 | 0.0018 | 0.000953 | 1.888772 |
| 1000 | 0.006311 | 0.003036 | 2.078722 | 1001 | 0.006853 | 0.003433 | 1.996213 |
| 2000 | 0.04043 | 0.014827 | 2.726782 | 2002 | 0.048672 | 0.021972 | 2.215183 |
| 4000 | 0.200552 | 0.069973 | 2.866134 | 3999 | 0.237666 | 0.086238 | 2.755931 |
| 8000 | 1.030997 | 0.339643 | 3.035531 | 8001 | 1.070841 | 0.369465 | 2.898356 |
| 16000 | 6.107547 | 1.939832 | 3.148493 | 16002 | 4.616019 | 1.720859 | 2.682392 |
| 32000 | 30.813303 | 9.363833 | 3.290672 | 32000 | 34.879671 | 10.877502 | 3.206588 |

Table 1 Comparison for the results obtained for the original SSE matrix x vector implementation and the arbitrary-size-matrix implementation

void matrix\_mult\_sse(int size, double \*vector\_in,

double \*matrix\_in, double \*vector\_out){

\_\_m128d a\_line, b\_line, r\_line;

int i, j, idx;

int size\_trunc = size - size%2;

for (i=0; i<size\_trunc; i+=2){

j = 0;

b\_line = \_mm\_load\_pd(&matrix\_in[i]);

a\_line = \_mm\_set1\_pd(vector\_in[j]);

r\_line = \_mm\_mul\_pd(a\_line, b\_line);

for (j=1; j<size; j++) {

b\_line = \_mm\_loadu\_pd(&matrix\_in[j\*size+i]);

a\_line = \_mm\_set1\_pd(vector\_in[j]);

r\_line = \_mm\_add\_pd(\_mm\_mul\_pd(a\_line, b\_line), r\_line);

}

for (j=1; j<size; j++) {

\_mm\_store\_pd(&vector\_out[i], r\_line);

}

}

if(size!=size\_trunc) {

j = 0;

b\_line = \_mm\_loadu\_pd(&matrix\_in[size\_trunc]);

a\_line = \_mm\_set1\_pd(vector\_in[j]);

r\_line = \_mm\_mul\_pd(a\_line, b\_line);

for (j=1; j<size; j++) {

b\_line = \_mm\_loadu\_pd(&matrix\_in[j\*size+size\_trunc]);

a\_line = \_mm\_set1\_pd(vector\_in[j]);

r\_line = \_mm\_add\_pd(\_mm\_mul\_pd(a\_line, b\_line), r\_line);

}

for (j=1; j<size; j++) {

double rest[2];

\_mm\_storeu\_pd(rest, r\_line);

vector\_out[size\_trunc] = rest[0];

}

}

}

Code snippet Function for SSE implementation for an arbitrary matrix size (not a multiple of the SIMD)

Figure . Optimized matrix size vs arbitrary matrix size comparison in terms of parallel speed-up over the sequential code

The implementation with the arbitrary N size (not multiple of 4) can be viewed in matrix-arbVal.c file and run via run\_sse\_arb\_float.sh. In Figure 2, it can be seen that the handling of corner cases of the arbitrary matrix size adds overhead to the execution (condition checks and/or branches inside the loops), reducing the speed-up over the sequential counterpart.

# Task 3

For this task we have assumed that the change should be done having the reference the code modified for Task 2. Therefore, for good means of comparison, the code has been adapted to perform computation on *double* type, for matrix x vector multiplication with arbitrary N size. The changes can be viewed in matrix-arbVal-double.c file and run via run\_sse\_arb\_double.sh.

We can observe that the speed-up is (as expected) twice as lower, because the level of parallelism decreases tot 2 FP instructions per core (as opposed to 4 for single-precision SSE). The best results are seen for matrix size of aprox 4000.

The last row (marked with red) shows significant performance degrade, when virtual memory is expanded into the disk (exceeding the RAM size because of the matrix size – 8 GB only for the input matrix). It seems that the parallel implementation is heavily impacted by the page switching mechanism between the RAM and the disk, therefore making this particular implementation worse than the sequential code. The algorithm may need to be improved (also using aligned data load would probably help), in order to reduce the overhead created by the memory access.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Double-precision floating-point implementation - arbitrary matrix size*** | | | |
| N (matrix / vector size) | Sequential (seconds) | SSE implementation (seconds) | Speed-up |
| 499 | 0.02475 | 0.0192 | 1.2890625 |
| 1001 | 0.011723 | 0.01016 | 1.15383858 |
| 2002 | 0.062154 | 0.059221 | 1.04952635 |
| 3999 | 0.293791 | 0.199865 | 1.46994721 |
| 8001 | 1.226782 | 0.855995 | 1.43316491 |
| 16002 | 5.808485 | 3.982106 | 1.45864651 |
| 32000 | 254.063376 | 378.784792 | 0.67073278 |

*Table 2 Results for in seconds and speed-up comparison for SSE double vs sequential execution for Task 3 (arbitrary matrix size)*

Figure 2. Speed-up sketch for double-precision floating-point SSE operation vs sequential code for different matrix size (arbitrary size)

# Task 4

The matrix times matrix implementation has as reference the implementation from Task 2 (vector times matrix – single-precision floating point), for means of computing efficiency, because double-precision would have doubled the compute time that is already complex.

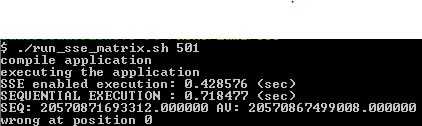


Figure 3 Sample result for matrix-matrix multiplication with N = 501

Interestingly, because the partial sums of the algorithm lead to intermediate results several times bigger than the dynamic range of the single-precision floating point representation, the program shows slight differences between the sequential code and the SSE implementation. This issue comes from the fact that the x87 FPU used in the SSE extension doesn’t use any additional guard bits (like the normal FPU in the pipeline which uses 80 bit FP representation for intermediate values), which can be confirmed by the fact that the numbers differ on their Least Significant Bit.

void matrix\_matrix\_mult\_sse(int size, float \*matrix\_1,

float \*matrix\_2, float \*matrix\_out){

int rows, cols;

int j, k;

\_\_m128 in1,in2,out;

float result[4];

int rest = size%4, size\_trunc = size - rest;

for(cols=0; cols<size; cols++)

for(rows=0; rows<size; rows++){

matrix\_out[rows\*size + cols] = 0.0;

out = \_mm\_set1\_ps(0.0);

for(j=0; j<size\_trunc; j+=4){

in1 = \_mm\_loadu\_ps(&matrix\_1[rows\*size+j]);

in2 = \_mm\_set\_ps(matrix\_2[cols+(j+3)\*size], matrix\_2[cols+(j+2)\*size], matrix\_2[cols+(j+1)\*size],matrix\_2[cols+(j)\*size]);

out = \_mm\_add\_ps(\_mm\_mul\_ps(in1, in2), out);

}

if(size!=size\_trunc) {

float aux[4] = {0}, aux2[4] = {0};

for(k=0; k<rest; k++) {

aux[k]=matrix\_1[rows\*size+j];

aux2[k]=matrix\_2[cols+(j++)\*size];

}

in1 = \_mm\_load\_ps(aux2);

in2 = \_mm\_set\_ps(aux[3], aux[2], aux[1], aux[0]);

out = \_mm\_add\_ps(\_mm\_mul\_ps(in1, in2), out);

}

\_mm\_storeu\_ps(result, out);

matrix\_out[rows\*size + cols] = result[0]+result[1]+result[2]+result[3];

}

}

Code snippet 2 SSE matrix-matrix multiplication function for aribitrary sized matrices with single-precision FP element representation

The function presented in *Code Snippet 2* is also included in matrix-matrix.c file that can be run via run\_sse\_matrix.sh.

# Task 5

For the purpose of further accelerating the matrix-matrix multiplication (with arbitrary matrix size), we have adapted the code for using the latest instruction set extensions from Intel, implemented in the 4th generation of Core processors (Haswell). The used extensions that came in handy were AVX2 and FMA. By these means, the application would have a theoretical throughput of 32 single-precision FLOP per cycle (8 FLOPs per core), with the biggest improvement coming from the fused-multiply-add block that accelerates the intermediate sums computation.

|  |  |  |  |
| --- | --- | --- | --- |
| ***Improvement of AVX2+FMA over SSE*** | | | |
| N (matrix size) | SSE implementation (seconds) | AVX2 + FMA implementation (seconds) | Speed-up |
| 399 | 0.232097 | 0.162034 | 1.4323969 |
| 501 | 0.477164 | 0.338361 | 1.41022163 |
| 602 | 0.861006 | 0.620024 | 1.3886656 |
| 703 | 1.392 | 1.000719 | 1.39099987 |
| 804 | 2.04744 | 1.498357 | 1.36645673 |

Table Comparison between the SSE implementation and AVX2 + FMA implementation

Figure Speed-up of the AVX2+FMA over the SSE implementation for a matrix-matrix multiply application

For assessing the implementation, we have compared it with the SSE code presented in ***Task 4***. The improvements are between 30-40 % (observable in *Figure 4*). We feel that the implementation can be further improved by increasing cache locality and, therefore, offloading the memory lane to the RAM, which seems to be the biggest bottleneck for this data-hungry application. The function is presented in *Code Snippet 3*.

Code snippet matrix\_matrix\_mult\_avx function which uses the new vector extensions (AVX2 + FMA)

void matrix\_matrix\_mult\_avx(int size, float \*matrix\_1,

float \*matrix\_2, float \*matrix\_out){

int rows, cols;

int j, k;

\_\_m256 in1,in2,out;

float result[8];

int rest = size%8, size\_trunc = size - rest;

for(cols=0; cols<size; cols++)

for(rows=0; rows<size; rows++){

matrix\_out[rows\*size + cols] = 0.0;

out = \_mm256\_set1\_ps(0.0);

for(j=0; j<size\_trunc; j+=8){

in1 = \_mm256\_loadu\_ps(&matrix\_1[rows\*size+j]);

in2 = \_mm256\_set\_ps(matrix\_2[cols+(j+7)\*size], matrix\_2[cols+(j+6)\*size], matrix\_2[cols+(j+5)\*size], matrix\_2[cols+(j+4)\*size], matrix\_2[cols+(j+3)\*size],matrix\_2[cols+(j+2)\*size],matrix\_2[cols+(j+1)\*size],matrix\_2[cols+(j)\*size]);

out = \_mm256\_fmadd\_ps(in1, in2, out);

}

if(size!=size\_trunc) {

float aux[8] = {0}, aux2[8] = {0};

for(k=0; k<rest; k++) {

aux[k]=matrix\_1[rows\*size+j];

aux2[k]=matrix\_2[cols+(j++)\*size];

}

in1 = \_mm256\_load\_ps(aux2);

in2 = \_mm256\_set\_ps(aux[7], aux[6], aux[5], aux[4], aux[3], aux[2], aux[1], aux[0]);

out = \_mm256\_fmadd\_ps(in1, in2, out);

}

\_mm256\_storeu\_ps(result, out);

matrix\_out[rows\*size + cols] = result[0] + result[1] + result[2] + result[3] + result[4]+result[5]+result[6]+result[7];

}

}