Motivation: A great portion of the multimedia applications such as template matching, object detection and face recognition involve intense work on searching, comparing and mathematical computation. Given the facts that high resolution images and HD videos are widely adapted, and enormous information is stored in the database as the source of recognition, multimedia applications need to be continuously accelerated to catch the pace of these changes. Besides the efforts spent by the algorithm designers, recent innovations in computer architecture together with their newly defined Instruction Set Architectures (ISA) give us the opportunity to accelerate the multimedia applications from the standpoint of hardware/software co-optimization.

The Intel SSE and AVX ISAs: The Intel SSE ISA has been developed since 1999, which has gone through 5 generations: SSE, SSE2, SSE3, SSE4.1, SSE4.2, SSE5. SSE5 was abandoned in 2007 since Intel decided to shift to the more promising AVX ISA. SSE, SSE2, SSE3 and SSE4.1 are generic ISAs which contain arithmetic operations, bitwise operations and memory-related operations. But SSE4.2 is a special ISA, in that it contains instructions mainly for the purpose of quick string search and comparison. It is suitable for applications like the Genome Sequence Comparison, file searching and template matching. The most recent SSE4 ISA uses the 128-bit XMM register which can store up to 16 "char" or 8 32bit "integer". For example, by using the SSE4.2 string and text ISA when searching for a certain character among 16 characters, only one instruction is needed instead of comparing with 1 character at a time and using 16 instructions to complete this task as what is normally implemented.

The Intel AVX ISA was first developed in 2008, then it has gone through 2 generations: AVX1 AND AVX2. AVX2 is an extension to the AVX1 by adding more instructions. The AVX ISA makes use of the 256-bit YMM registers. AVX contains instructions mainly for floating point operations. The YMM register can contain up to 4 double-precision floating points or 8 single-precision floating points. The instructions inside the AVX are general purposed instructions, such as the arithmetic operations and the bitwise operations like the SSE ISA. However, AVX adds more fine-grained memory-related instructions to better arrange the order of data stored inside the YMM register.

Note that there are other new ISAs developed by Intel for special purposes, such as the new Fused-Multiply-Add ISA. However, since the machines used in this project are not up-to-date enough to support these instructions, we just skip these instructions even though they may lead to a better speedup.

The Intel Opencv Library: The OpenCV (Open Source Computer Vision) is a library of programming functions for real time computer vision. It contains many optimized functions that cover different multimedia applications ranging from image processing to machine learning. The source codes are well written in multiple computer languages. For the concern of performance, this project will work on modifying the Cpp source code. Due to the inability for live interaction through camera, the applications selected for acceleration in this project include image processing and object detection. Due to the large ISA set size, we are not going through each instruction in detail. Readers can go through the following links to get the most updated explanation about each instruction and its usage. Instead, we are going to explain each used instruction on the fly as we go through 4 case studies of the core functions extracted from three different multimedia applications.
Case Studies:

The three applications we considered for acceleration are finding an object (/samples/c/find_object.c), generation of colors (/samples/cpp/gencolors.cpp) and latent SVM detector (/samples/cpp/latentsvm_multidetect.cpp). More details about these applications can be found through the online resources. There are four sub-functions that are identified to be the most time consuming portion when running the three applications. The first one is the compareSURFDescriptors. The original code and modified code are listed as follows.

Case 1:

Original:

```c
for( int i = 0; i < length; i += 4 ){
    double t0 = d1[i] - d2[i];
    double t1 = d1[i+1] - d2[i+1];
    double t2 = d1[i+2] - d2[i+2];
    double t3 = d1[i+3] - d2[i+3];
    total_cost += t0*t0 + t1*t1
                + t2*t2 + t3*t3;
}
```

Modified:

```c
int chunk = length / 4;
for(i = 0; i < chunk; i++){
    __m128 m0, m1;
    m0 = _mm_load_ps(&d1[4 * i]);
    m1 = _mm_load_ps(&d2[4 * i]);
    m1 = _mm_sub_ps(m0, m1);
    m1 = _mm_add_ps(m1, m1);
    m2 = _mm_shuffle_ps(m1, m1, _MM_SHUFFLE(2,3,0,1));
    m1 = _mm_add_ps(m1, m2);
    total_cost += ((float*)&m1)[0];
    if( total_cost > best )
        break;
}
```

Basically this function tries to compute the difference between the corresponding positions of two floating point arrays. Then the differences will be squared and summed up. When using the AVX instructions to modify the functions, the _mm_sub_ps and _mm_mul_ps each can compute 4 floating points operation in parallel. Then the final solution is given by doing one horizontal addition _mm_hadd_ps, one shuffle operation (to move the temp results to the first position inside __m128 register), and one final addition _mm_add_ps (to store the results in the first position of the __m128 register). This case study is relatively straight forward and I'm not going into more details.
Case 2:

Case 2 refers to the minmaxloc function used in the find_object.c. Basically it will find the locations of the minimum and maximum numbers inside a floating point array of size N while also returning the minimum and maximum values. The results of this case study shows that for AVX, the comparison operation is always more costly than the min/max operations. The comparison instructions in AVX are also supposed to be slower than the comparison instructions in the SSE. This case study also suggests that the applications with simple task to perform have more chance to be accelerated by the Intel ISA. This is because multiple tasks always require unnecessary loop repetition since they cannot be packed and accomplished inside one loop.

Original:

```c
float minval = FLT_MAX, maxval = -FLT_MAX;
for (i = 0; i < N; i++, ++it ){
    float v = *(const float*)it.ptr;
    if( v < minval ){
        minval = v;
        minidx = it.node()->idx;
    }
    if( v > maxval ){
        maxval = v;
        maxidx = it.node()->idx;
    }
}
if( _minval )
    *__minval = minval;
if( _maxval )
    *__maxval = maxval;
```

Modified:

```c
__mm128 m0, m1, m2, m3, m4, minArray, maxArray;
int chunk = N / 4;
for(i = 1; i < chunk; i++){
    m0 = __mm_load_ps( (const float*)it.ptr );
    it += 4;
    m1 = __mm_min_ps(m0, minArray);
    m2 = __mm_max_ps(m0, maxArray);
    m3 = __mm_cmp_ps(m0, minArray, _CMP_LT_OS);
    m4 = __mm_cmp_ps(m0, maxArray, _CMP_GT_OS);
    int* mask1 = (int*) &m3;
    int* mask2 = (int*) &m4;
    for(int j = 0; j < 4; j++){
        if(mask1[j] == -1)
            minPos[j] = 4 * i + j;
        if(mask2[j] == -1)
            maxPos[j] = 4 * i + j;
```
The key instructions used here are the _mm_min_ps, _mm_max_ps and _mm_cmp_ps. Basically, the min/max position will select the min/max elements inside each corresponding position of the two source registers and store them into the destination register. The compare instruction will return the comparison flag of each corresponding position and store them into the destination register. _CMP_SIGN_OS is the Intel AVX pre-defined macro that defines which kind of comparison to perform. For this particular example, since we are performing the searching for the minimum and maximum values of a floating point array and at the same time finding the corresponding position, we eventually find that the modified code actually runs slower than the original code as shown later in the simulation results section. However, if we are only performing one task (only finding the minimum and maximum value), the modified code can have a significant speedup.

**Case 3:**

The third case refers to the convertPoints function. Basically, it will try to perform conversions to two arrays of points where point is a structure that contains two integer values. This example shows that loading and operating on structures can be a really time-consuming task for AVX ISA. The data after loading are interleaved like X0Y0X1Y1X2Y2X3Y3. The later operations would require us to arrange the memory as X0X1X2X3Y0Y1Y2Y3 for fast runtime. This would require a long processing time of arranging the contents inside the XMM registers, which involve a series of permute, shuffle and blend instructions. Also note that based on the feature of the structures and the operations that are performed on these structures, there can be different combinations of permute, shuffle and blend instructions to achieve the same goal. However, the speedup may be quite different from one strategy and another. The code snippet bellow has detailed comments indicating how the contents locations vary inside the XMM register as the procedure moves on.

**Original:**

```c
for (i = 0; i < kPoints; i++){
    // scaling factor for root filter
    scale = 8 * powf(step, (float)(levels[i] - initialImageLevel));
    points[i].x = (int)((points[i].x - bx + 1) * scale);
    points[i].y = (int)((points[i].y - by + 1) * scale);

    // scaling factor for part filters
    scale = 8 * powf(step, (float)(levels[i] - lambda - initialImageLevel));
    for (j = 0; j < n; j++){
        partsDisplacement[i][j].x = (int)((partsDisplacement[i][j].x - 2 * bx + 1) * scale);
        partsDisplacement[i][j].y = (int)((partsDisplacement[i][j].y - 2 * by + 1) * scale);
    }
}
```

**Modified:**
for (i = 0; i < kPoints; i++){
    scale[i] = 8 * powf(step, (float)(levels[i] - initialImageLevel));
    scale2[i] = 8 * powf(step, (float)(levels[i] - lambda - initialImageLevel));
}

for(i = 0; i < outer_chunk; i++){
    __m128 m0 = _mm_load_ps(&scale[4 * i]);

    __m256i temp = _mm256_load_si256((__m256i*) &points[4 * i]);
    __m256 temp2 = _mm256_cvtepi32_ps(temp);

    // initially: x0y0x1y1x2y2x3y3
    // x0x1y0y1y2y3x2x3
    v4si mask1 = {9,8,8,9};
    __m256 temp3 = _mm256_permutevar_ps(temp2, mask1);
    // y2y3x2x30x1x0y1
    __m256 temp4 = _mm256_permute2f128_ps(temp3, temp3, 0x01);
    // x0x12x3y0y1y2y3
    temp3 = _mm256_blend_ps(temp3, temp4, 0b00110011);
    // x0x1x2x3y2y3y0y1
    v4si mask2 = {0xd,4,4,0xd};
    temp3 = _mm256_permutevar_ps(temp2, mask2);

    // x0x1x2x3
    __m128 m1 = _mm256_extractf128_ps(temp3, 1);
    // y2y3x0y1
    __m128 m2 = _mm256_extractf128_ps(temp3, 0);

    m1 = _mm_mul_ps(m1, m0);
    m2 = _mm_mul_ps(m2, m0);

    __m256 temp5 = _mm256_insertf128_ps(temp5, m1, 0);
    temp5 = _mm256_insertf128_ps(temp5, m2, 1);
    __m256i m3 = _mm256_cvtps_epi32(temp5);

    int* mask = (int*) &m3;
    for(j = 0; j < 4; j++)
        points[4 * i + j].x = mask[j];

    for(; j < 8; j++)
        points[4 * i + j - 4].y = mask[j];
}

Case 4:

Case 4 refers to the clipping box function. Similar to case 3, it also operates on an array of structures, where the structure type is also point. However, the operations in case 4 is much simpler than in case 3.

Original:
for (i = 0; i < kPoints; i++)
    if (points[i].x > width - 1){
        points[i].x = width - 1;
    }
    if (points[i].x < 0){
        points[i].x = 0;
    }
    if (points[i].y > height - 1){
        points[i].y = height - 1;
    }
    if (points[i].y < 0){
        points[i].y = 0;
    }
}

Modified:

int chunk = kPoints / 4;
for (i = 0; i < chunk; i++)
{
    __m128 m0 = _mm_load_ps(&scale[4 * i]);
    __m256i temp = _mm256_load_si256((__m256i*) &points[4 * i]);
    __m256 temp2 = _mm256_cvtepi32_ps(temp);
    // initially: x0x1x2x3y0y1y2y3
    // x0x1y0y1y2y3x2x3
    v4si mask1 = {9,8,8,9};
    __m256 temp3 = _mm256_permutevar_ps(temp2, mask1);
    // y2y3x2x3x0y1y0y1
    __m256 temp4 = _mm256_permute2f128_ps(temp3, temp3, 0x01);
    // x0x1x2x3y2y3y0y1
    temp3 = _mm256_blend_ps(temp3, temp4, 0b00110011);
    // x0x1x2x3y0y1y2y3
    v4si mask2 = {0xd,4,4,0xd};
    temp3 = _mm256_permutevar_ps(temp2, mask2);
    float e1 = (float)(width - 1);
    float e2 = (float)(height - 1);
    __m256 temp5 = _mm256_set_ps(e1, e1, e1, e1, e2, e2, e2, e2);
    __m256 temp6 = _mm256_setzero_ps();
    __m256 m3 = _mm256_cmp_ps(temp3, temp5, _CMP_LE_OS);
    __m256 m4 = _mm256_cmp_ps(m3, temp6, _CMP_GE_OS);
    temp = _mm256_cvttps_epi32(m4);
    int* mask = (int*)&temp;
    for(j = 0; j < 4; j++)
        points[4 * i + j].x = mask[j];
    for(; j < 8; j++)
Simulation Setup:

The experiments are running on a quad-core core-i7 machine with 3.4Ghz CPU and 12 Gigabytes memory. The ISA supported by this machine are circled out in red rectangular boxes shown in the following picture. As you can see, the machine supports all SSE ISAs but with only AVX1.

The Opencv library is version 2.4.5 which is the most up-to-date version. It is not easy to work with since there is not a thorough documentation about how the files are arranged. In this case, I made a video and put it online to describe how to quickly pin-point a file in the Opencv library and how to successfully compile the code modified using the Intel ISAs under the configuration of the library. Here are the links:

https://www.youtube.com/watch?v=ISap9zEGE2I
https://www.youtube.com/watch?v=pqSgT0quMBc

For compilation, the optimization level is -O and all SSE and AVX flags are includes as follows:

CFLAG = -g -Wall -mavx -msse -msse2 -msse3 -msse4 -O

Simulation Results:

All functions are tested to be correct and now we want to show the speedup from the AVX instructions. In the figure below, CSD refers to case 1. MML refers to case 2. CNVP refers to case 3 and CLPB refers to case 4. The runtime of the original code is always normalized to 1. As can be seen in the figure, case 1 and case 3 achieves speedup while for case 2 and 4, the modified codes run slower. For case 2, the reason is that we are performing multiple tasks inside the “for” loop and most of the runtime overhead comes from the time consuming comparison operation. For case 4, the runtime overhead also comes from the costly comparison operation, since we need to go through a “for” loop to extract the comparison results stored inside the register.
Conclusion and Future Work:

In this work, the Intel AVX ISA is applied to accelerate the OpenCV Library. Simulation results show that not all applications get accelerated when the Intel ISA is applied. This is either because the multiple tasks inside each application make a generic acceleration strategy of accelerating multiple tasks at the same time hard to realized, or because of the extra effort spent on rearranging the contents inside the XMM registers.

We also found some observations that are good for automated code compilation based on the new ISA. For example, we can encode the instruction sequence for arranging the data inside the XMM register into the compiler, since such kinds of arrangement are frequently encountered in practice. But note that with the development of the ISA and the addition of new instructions, the current solution of arranging the data inside the register may not be the optimal anymore. In that case, new solution should be developed.