A Fast Feature Extraction in Object Recognition Using Parallel processing on CPU and GPU

Junghul Kim, Eunsoo Park, Xuenan Cui and Hakil Kim
School of Information Engineering,
Inha University
253, Yonghyun-dong, Nam-ku, Incheon, Rep. Korea
E-mail: {jekim, espark, xncui, hikim}@vision.inha.ac.kr

Abstract—Due to the advents of multi-core CPU and GPU, various parallel processing techniques have been widely applied to many application fields including computer vision. This paper presents a parallel processing technique for real time feature extraction in object recognition by autonomous mobile robots, which utilizes both CPU and GPU by combining OpenMP, SSE (Streaming SIMD Extension) and CUDA programming. Firstly, the algorithms and codes for feature extraction are optimized and implemented in parallel processing. After the parallel algorithms are assured to maintain the same level of performance, the process for extracting key points and obtaining dominant orientation with respect to the key points is parallelized. Following the extraction is the construction of a parallel descriptor via SSE instructions. Finally, the GPU version of SIFT is also implemented using CUDA. The experiments have shown that the CPU version of SIFT is almost five times faster than the original SIFT while maintaining robust performance. Further, the GPU-Parallel descriptor achieves acceleration up to five times higher than the CPU-Parallel descriptor at a cost of a bit lower performance.

Keywords—Parallel processing, OpenMP, SSE, CUDA, Feature Extraction, SIFT, SURF

I. INTRODUCTION

The process of recognizing objects can be divided into the following three stages. In the first stage, the necessary features are represented via global or local information. In the second stage, an optimal decision rule for classifying the data is designed on the basis of the extracted features [1]. The third stage consists of matching and recognizing the new data. To enhance performance, representation, learning, and recognition processes have been treated in an integrated fashion rather than independently [2-5].

Among feature-based approaches, the Scale Invariant Feature Transform [2] algorithm has been demonstrated to have good performance with respect to variations in the size, rotation, and translation of the image. Mikolajczyk [7, 8] has shown that methods based on the SIFT algorithm can also achieve robust performance. Furthermore, to enhance performance and accelerate processing, various algorithms have been proposed, including PCA-SIFT [3] and SURF (Speed up Robust Features) [4]. Because constructing and matching descriptors in these SIFT-based methods involves a higher-dimensional descriptor, they are very time-consuming and difficult to apply for real-time processing. Therefore, a stable high-speed algorithm is needed [8] to achieve real-time recognition for mobile robots.

II. PARALLEL PROCESSING USING OPENMP, SSE AND CUDA

A. Parallel processing configuration using OpenMP

OpenMP is an API (Application Program Interface) used to generate multiple threads for parallel programming in a public, shared memory environment. It consists of three components: compiler directives, a run-time library and environmental variables [9]. Inserting the directive into a program can be processed in parallel into a loop, but the OpenMP compiler does not automatically support analysis of all issues and parallelization. Synchronization, data dependency, and other issues caused by parallelization should be directly handled by pre-analysis of the sequential program [10].

B. Parallel processing configuration using SSE

The MMX instruction set was initially introduced in the Intel Pentium II processor generation, and then SSE, SSE2, SSE3, and SSSE3 (Supplemental SSE3) [11, 12] instruction sets which were extended versions of MMX as provided in the Intel Pentium 4 and Core 2 Duo processor generations. MMX instructions can process eight 8-bit integers at the same time via a 64-bit register, and SSE can process four SIMD 32-bit floating point instructions via a 128-bit register, as shown in Fig. 1. SSE2 includes SIMD instructions that can process
sixteen 8-bit integer data types and can handle two double-precision data types at the same time. In the 90 nm processor technology, the SSE3 instruction set includes 13 additional instructions for SSE and SSE2 to enhance the capabilities of x87-floating point processors. In addition to enhancing the functions of SIMD integer arithmetic, SSSE3 includes 32 additional instructions and SSE4 includes 54 additional instructions that were introduced in the Intel 64 bit processor via 45 nm technologies.

These SIMD instructions can be executed in Inline Assembler, Automatic Vectorization, or Intrinsic functions [16, 17]. Inline Assembler directly uses SSE instructions to accelerate processing, but it is difficult to program and unfeasible in practice due to structural dependency. Automatic Vectorization of the Intel C/C++ compiler improves the performance via a loop, which is automatically implemented by SSE and is simple from the perspective of programming. Meanwhile, Intrinsic functions lie somewhere between Inline Assembler and Auto Vectorization, and enable easier utilization of SSE instructions that are supported in Intel. The use of intrinsic functions is suitable for all Intel processors that support the SIMD compiler, and can achieve a performance comparable to Assembler. Therefore, the proposed method utilizes intrinsic functions for more effective high-speed object recognition.

C. Parallel processing configuration using CUDA

GPU is a special-purpose processing unit that is designed to resolve the bottlenecks caused by Graphics applications. It has been extensively studied in the field of graphics and has been used as a general-purpose processing device due to its higher transistor densities and SIMD parallel hardware structure [13]. This technology is generally called GP-GPU (General Purpose Computation on GPUs) and is used to assure real-time performance in applications related to video encoding or computer vision that need to handle large amounts of data [14, 15].

CUDA is designed for general-purpose computing on the GPU hardware and software architecture using C. NVIDIA created and distributed a development toolkit of drivers and software that can be used in high-level graphical devices such as GeForce 8, Quadro NVS 130M and Tesla. CUDA considered GPU hardware as an independent platform that can provide a programming environment and minimize the need for understanding the graphics pipeline. Also, the CPU and the GPU can be used for developing heterogeneous systems [13].

The process of feature extraction using CUDA is performed by the model shown in Fig. 2. When the kernel function to be executed with CUDA is ready, a grid composed of one block must be configured. The block generates a large number of threads to share data with other threads and then parallel processing can be performed. The kernel will be performed by the thread that depends on the configuration that can affect the execution speed.

III. PARALLEL FEATURE EXTRACTION

When compared to Harris-Laplace feature extraction, the Hessian-Laplace method is similar to DoG (Difference of Gaussian) but it is simpler due to utilization of the Gaussian filtered image [2] and also has strong performance. We extract key points based on the Hessian and ensure stability by applying a box filter [4]. First, the process of extracting features involves selecting candidate key-points based on a Hessian matrix in the scale space and then a Taylor expansion is utilized to obtain a more accurate location.

A. Parallel scale space configuration

For extracting candidate key-points using the Hessian determinant of the image, a scale space image \( L(x, y, \sigma) \) is computed by the convolution of a variable-scale Gaussian \( G(x, y, \sigma) \) over the input image

\[
L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{1}
\]

where \( L(x, y, \sigma) \) can be regarded as a Gaussian-blurred image with a different level of variance. This operation is invariant with respect to scale changes of the image and is achieved by searching for stable features across all possible scales.

Data dependency in the scale space is reduced by refactoring code and relocating loop invariant code for parallelization. Fig. 3 shows the configuration of parallel scale space images. Given the number of variable scales \( \sigma_1, \ldots, \sigma_m \) for Gaussian operation, the same number of threads are required for constructing individual pyramids of \( m \) octaves simultaneously. The initial image in each pyramid is smoothed...
with a different scale of Gaussian and subsequently sub-sampled in parallel. While the values of \( n \) and \( m \) are independent, if at least one of them increases, a trade-off between the efficiency and computational time is required \([2]\).

Moreover, the number of threads can be larger than the number of processors, however, in that case, unassigned threads must wait for the cores to complete the assigned threads which increases overhead. In our experiments, the number of pyramids \( n \) is four and \( m \) is also four.

**B. Parallel detection of key points**

In the scale space images, the Hessian matrix \( \mathbf{H}(x, y, \sigma) \) is computed from the second-order derivative of the Gaussian blurred image.

\[
\mathbf{H}(x, y, \sigma) = \begin{bmatrix}
L_{xx}(x, y, \sigma) & L_{xy}(x, y, \sigma) \\
L_{xy}(x, y, \sigma) & L_{yy}(x, y, \sigma)
\end{bmatrix}
\]  

where \( L_{xy} \) is the second-order derivative of the Gaussian image, horizontally with respect to the \( x \)-direction and vertically with respect to the \( y \)-direction. In general, Gaussian blurring is optimal for scale-space analysis. In practice, the Gaussian filter is discretized and cropped, and then a box filter as designed in SURF \([4]\) is applied to the resulting image after the second derivative in order to reduce the number of false candidate points.

The 9×9 box filters in Fig. 4(a) are approximations of the LoG (Laplacian of Gaussian) with \( \sigma = 1.2 \). They represent the lowest scale (i.e., highest spatial resolution) for computing the blob response maps. The size of the box filter increases according to the octave of the input image. Because the scalability of the Gaussian and the box filter sizes are fixed, filtered responses denoted as \( D_{xx} \), \( D_{yy} \) and \( D_{xy} \) can be approximated in advance.

For the input image in Fig. 4, the number of octaves represents the order in one pyramid, and the pyramid number also indicates the order of the space that is shown in Fig. 3. The process of approximation is split into \( k \) threads and computed in parallel. As the number of threads \( k \) increases, the overhead time also increases because the determinant should be calculated after all threads have been completed. In this experiment, \( k \) is set to four.

![Figure 3. Configuration of parallel scale space images.](image)

**C. Construction of descriptors using SSE instructions**

1) Parallel extraction of orientation

In order to achieve robustness with respect to rotation of objects in the images, a dominant orientation is needed with respect to each key point that will be used to construct the descriptor. To obtain the orientation, Haar wavelet responses in the \( x \) and \( y \) directions are calculated as shown in Fig. 5(a), where the size of the wavelet is \( 4s \), the calculated area is circular with a radius of \( 6s \) around the key point, and \( s \) is the
scale of each key point in this step. This process again uses the integral image and is split into individual threads in a manner similar to box filtering. Thus, the wavelet response is calculated using only six additional operations and is performed simultaneously in different circular areas using OpenMP.

![Figure 5. Procedure to obtain the orientation and ROI](image)

After the wavelet responses are obtained they are represented as vectors. All responses within the rotating orientation window are summed to generate an orientation histogram. The histogram has 36 bins covering 360°. The longest vector in the orientation histogram corresponds to the dominant orientation of the local area. The arrows in Fig. 5 (b) show the dominant orientation at each stored key point.

2) Parallel construction of descriptors using SSE instructions

To construct the descriptor, we construct a square ROI which is rotated with respect to the orientation at each key point as shown in Fig. 5 (b). The ROI area of 16×16 is partitioned into 16 sub-regions, each having a 4×4 window. The size of the ROI can be empirically determined, however, 16×16 is sufficient for SSE instructions because the process can be accelerated when the data are aligned to be multiples of 16. Then, the wavelet responses (horizontal and vertical responses, $dx$ and $dy$) are computed in the $x$ and $y$ directions by the same means as extraction of the orientation.

For each sub-region, the directional responses $dx$ and $dy$ are summed into $\Sigma dx$ and $\Sigma dy$, respectively, and the sum of absolute values of the responses is also obtained. Therefore, a descriptor is constructed at each sub-region as $[\Sigma dx, \Sigma dy, \Sigma |dx|, \Sigma |dy|]$, and the descriptor of each ROI consists of 16 vectors from 16 sub-regions. Hence, the length of the descriptor for each ROI is 64. All these processes are implemented by SSE instructions over 128-bit registers as shown in Fig. 6.

![Figure 6. Parallel process for constructing a descriptor for a sub-region.](image)

Initially, in order to compute the sum of responses, $dx$ of the first sub-region in an ROI is loaded via _mm_load_si128 and then the absolute value $|dx|$ is computed via _mm_abs_epi32. Basically, since the data type of the wavelet response is floating point (16 bits), four double-precision floating point (32 bits) data are loaded in a 128-bit register at the same time. As shown in Fig. 6 after all $dx$ values are loaded in four registers, each column is summed in parallel via the _mm_add_epi32 function and stored in $T_x$. At the same time, the sums of $|dx|$, $dy$, and $|dy|$ are computed in parallel and stored in the same register $S_y$ along with others. Finally, the register $S_y$ represents the descriptor of a sub-region.

After 16 descriptors for 16 sub-regions in a ROI are obtained, the descriptor vector of 64 dimensions is normalized to achieve invariance to contrast [4]. Fig. 7 and 8 describe the parallel processes of computing the $l^2$-norm of an ROI descriptor vector and dividing the ROI descriptor vector by the $l^2$-norm of the vector $|x| = \sqrt{\sum_{i=1}^{n} x_i^2}$, i.e., $x = [x_1, x_2, ..., x_n]$ where $n = 64$. To compute the $l^2$-norm, the sum of the squared individual components is needed.

![Figure 7. Parallel process for computing the $l^2$-norm at each ROI area.](image)

The squared individual components can be obtained via _mm_mul_ps. This process is iterated for computing $M_1, M_2, ..., M_n$ as shown in Fig. 7, and then used to determine the sum for the components $\Sigma dx, \Sigma dy, \Sigma |dx|, \Sigma |dy|$ of the ROI. In order to generate four $l^2$-norms of the ROI area in parallel, a shuffle operation _mm_shuffle_ps, an addition operation _mm_add_epi32, a shuffle operation _mm_shuffle_ps, an addition operation _mm_add_epi32, and finally a square-root operation are applied in series to $[\Sigma dx, \Sigma dy, \Sigma |dx|, \Sigma |dy|]$. 

3945
In Fig. 8, four $l^2$-norms are used to construct a unit vector by dividing 4-dimensional descriptors of $n$ sub-regions in parallel with `_mm_div_ps` and `_mm_mul_ps`. The final result represents the normalized 64-dimensional descriptor vector for each key point. These procedures are iterated for every ROI area corresponding to each key point as shown in Fig. 5(b). After completion, the descriptor is stored as a register type by `_mm_store_ps`.

After extracting the key points, the histogram of orientation is obtained using the variation of gradient around the key points and the maximum angle in the histogram is stored as the elements of that point’s angle. The process of finding the maximum value through the histogram by comparison with the shared value in the ROI area is similar to the third stage. Based on the final orientation, the rotated area is constructed within the blurred image which is on the texture memory. The descriptor is constructed by summing the gradient in eight directions. The number of block is set to the number of key points in order to simplify the operation and the thread number in each block is set to 16.

V. EXPERIMENTAL RESULTS

A performance comparison has been conducted between the proposed and existing methods based on the CPU and GPU. The detector and descriptor are evaluated with respect to an 800×640 graffiti [7] image using an Intel Core 2 Duo 2.66 GHz processor in a Windows XP operating system and Nvidia Geforce 8800 GT graphics card. Fig. 10 shows a comparison of computational time in key point detection for Harris-Laplace, Hessian-Laplace, Harris-Affine, Hessian-Affine, and the proposed Parallel-Hessian with respect to the image size. In this experiment, the Parallel-Hessian is about 2.5 times faster than Hessian-Laplace which is the fastest among the other methods.

We also compared the performance of the proposed parallel descriptor against SIFT [6], PCA-SIFT [9] and SURF [10] that are based on a Hessian detector with the same conditions. The processing time for descriptor construction includes the key point extraction. As shown in Fig. 11, we found that the Parallel descriptor on the CPU via OpenMP and SSE is 4.5
times faster than the CPU descriptors. The GPU-parallel method using CUDA outperformed the other CPU methods as well as CPU-parallel method; it achieves acceleration up to 5 times and can extract 1100 features at an average frame rate of 15 Hz.

In addition, the performance of the proposed methods was evaluated in terms of the recall versus 1-precision as represented in Mikolajczyk [6, 7]. An ideal descriptor provides a recall of one for any precision. However, in practice descriptors are affected by several factors so that the recall slowly increases as the 1-precision is increased. Fig. 12 shows the corresponding curves for various descriptors and the proposed Parallel descriptor. The performance of the proposed CPU-Parallel method is comparable to other descriptors, but CUDA-SIFT has decreased performance caused by trade-off in speed.

Figure 12. Performance of descriptor respect to recall versus 1-precision curve.

VI. CONCLUSIONS

This paper has presented new parallel processing schemes using OpenMP and SSE and CUDA for fast feature extraction in object recognition with autonomous mobile robots. A parallel descriptor is achieved by analyzing, optimizing, and refactoring data from the original algorithms to be suitable for parallel processing. The proposed detector and descriptor are compared with other methods. Our proposed GPU-parallel method is shown to be faster than other methods including CPU-parallel method, but the performance is somewhat less. The CPU-parallel method using OpenMP and SSE, however, is slower than the CUDA-based method but faster than the original SIFT while maintaining robust performance.

The proposed fast feature extraction method represents an important advantage for recognition of complex and dynamic environments in realtime and it can be applied to various image processing applications. However, the proposed GPU method can result in additional overhead for searching and matching.

ACKNOWLEDGMENT

This work was supported by the Korea Science and Engineering Foundation (KOSEF) under grant NO. 3148201068

REFERENCES