High Speed Implementation of Segmentation by PSPNet on a Latest CPU

Junya Morioka[1] and Ryusuke Miyamoto[2]

Abstract— **Semantic segmentation, a fundamental task in image processing, is applied to various significant applications such as visual navigation of a robot, where accurate results and high speed processing are required. GPUs play an important role in high speed implementation but other options should be shown because various heavy procedures run in the process of the visual navigation. In this study, we propose a novel method to improve the calculation speed of PSPNet on the latest AMD Ryzen 9950X CPU. First, in order to improve the calculation speed of the Convolution layer, which accounts for the majority of PSPNet calculations, we utilize SIMD instructions (AVX-512), multi-core parallelization, and L1, L2, and L3 caches to accelerate matrix multiplication calculations using im2col. Next, we improve the calculation speed of the entire model by fusing the Convolution layer and the Batch Normalization layer. Compared with the PyTorch implementation, the proposed method achieved more than 79% of the peak performance in matrix multiplication while maintaining accuracy and succeeded in speeding up inference time by 47.712% on the Cityscapes dataset and 39.096% on the Visual Navigation dataset.**

I. Introduction

Semantic segmentation is a task to assign a class label to all pixels. To solve this task, various methods have been proposed for a long time including traditional approaches such as Meanshift [1], Levelset [2], MRF [3], CRF [4], Graphcut [5] etc. However, it was difficult to achieve practical accuracy before the emergence of deep learning similar to many other computer vision tasks except for the CoF [6], which showed remarkable performance on limited images presented at the same conference as PSPNet [7], one of the most practical methods for semantic segmentation. In recent years, the inference accuracy of semantic segmentation has become better gradually [8] by the introduction of the Transformer architecture that showed significant results in the field of natural language processing.

Since semantic segmentation is an essential task in image processing and computer vision, much useful information for other various tasks can be obtained when segmentation is solved. Especially, for depth estimation, where semantic boundaries in an input image are important, some methods try to improve the estimation accuracy by sharing features with segmentation for the training process [9]. Considering applications of semantic segmentation, some methods try to directly apply the results of semantic segmentation, such as 3D structure estimation of a room [10] and visual navigation of a robot [11]–[15].

Focusing on the visual navigation of a robot that is a quite important application in terms of development of human society, it is important to obtain accurate segmentation results in real-time. Our previous results showed that appropriate datasets can improve the inference accuracy of semantic segmentation for a target application. Regarding processing speed, one of the latest gaming laptops having a GPU enables real-time processing of semantic segmentation when the moving speed of a robot is limited to the walking speed of humans, which is sufficient for the human-robot coexisting environment. However, further efficient implementation is indispensable for visual navigation because several kinds of procedures must be processed in real-time; in addition to the improvement of total performance, the power consumption is expected to be reduced.

It is obvious that the recent drastic advances of several kinds of deep learning approaches are heavily supported by the performance improvement of GPUs. However, we can also appropriately exploit the computation power of CPUs that have become more powerful: higher processing frequency and increased computing units in a processor. Furthermore, CPUs have the advantage of not requiring data transfer between main memory and video memory, which is necessary for GPUs. Therefore, this paper proposes a high speed implementation of semantic segmentation by PSPNet to demonstrate the effectiveness of utilizing CPU computing resources, which remain unused in many current implementations. In our implementation, parallel processing using single instruction multiple data (SIMD) instructions on multiple cores and L1, L2, and L3 cache optimization are applied. To show the effectiveness of the proposed method, experiments using datasets created for training a classifier for visual navigation are conducted.

II. Related Work

A. Semantic Segmentation

Two representative models in semantic segmentation are PSPNet [7], which is based on Convolutional Neural Networks (CNNs), and SegFormer [8], which is based on the Transformer architecture. In our research group, these are applied to autonomous driving and their effectiveness in real environments is being verified [11]–[15]. Compared with the CNN-based PSPNet, the Transformer-based SegFormer has a higher computational cost and is difficult to use for real-time

*This work was supported by JSPS KAKENHI Grant Number JP25K01236

[1]Junya Morioka is with Department of Computer Science, Graduate School of Science and Technology, Meiji University, Kawasaki, Japan mjun@cs.meiji.ac.jp

[2]Ryusuke Miyamoto is with Department of Computer Science, School of Science and Technology, Meiji University, Kawasaki, Japan miya@cs.meiji.ac.jp

TABLE I
LAYER STRUCTURE OF PSPNET(RESNET50)

Layer Name	Number of Layers
Convolution (7x7)	1
Convolution (3x3)	21
Convolution (1x1)	37
Batch Normalization	58
ReLU	54
MaxPooling	1
AvgPooling	4
ArgMax	1

TABLE II
SPECIFICATION OF AMD RYZEN 9 9950X

Architecture	Zen5
Base Clock	4.3GHz
Boost Clock	5.7GHz
Number of Cores	16
Number of Threads	32
L1 Cache(per core)	80KiB
L2 Cache(per core)	1MiB
L3 Cache(shared)	64MiB
Supported SIMD Instructions	AVX2, AVX-512, FMA3
Floating Point Operation Units(FMA)	2

inference. Therefore, in this study, with the application to visual navigation in mind, we focus on speeding up PSPNet.

B. Acceleration Techniques for Convolution Operations

The majority of PSPNet calculations are convolution operations. Therefore, speeding up convolution operations is the most effective way to speed up PSPNet. There are three main methods for accelerating convolution operations.

The first method is im2col [16]. im2col accelerates CNN computation by reducing CNN convolution operations into matrix multiplications between two 2D matrices. This method can be applied to CNNs with all filter sizes and is most widely used because it can be easily implemented using fast matrix calculation libraries such as BLAS [17].

The second method is called Direct Convolution [18]. This method converts the format of the input, filter, and output matrices to a format suitable for SIMD instructions like NCHW8c or NCHW16c, and then performs convolution operations. In the case of FP32 data type, this method is only applicable when the number of channels of the input matrix is a multiple of 8 or 16.

The third method is Winograd [19]. This method significantly reduces the number of multiplications in convolution operations using only small-sized filters. In the case of convolution operations on GPUs, due to the slow processing of multiplications compared to additions, the effect of speeding up convolution operations using Winograd is significant. In modern CPUs, there is no significant difference in the speed of addition and multiplication, so the effect of speeding up convolution operations is limited. In addition, it is known that Winograd has a problem with cumulative calculation errors [20].

In this study, we adopt the im2col method, which is applicable to all CNNs, because our target is PSPNet, which contains CNNs with various filter sizes on CPU.

III. METHODS OF ACCELERATING PSPNET ON CPU

This section describes methods for accelerating PSPNet on CPU. In the following explanation, the input and output matrices are treated as 4-dimensional matrices, represented in NCHW format where N is the batch size, C is the number of channels, H is the height, and W is the width, respectively.

A. Acceleration of Convolution Operation on CPU

This section presents methods for accelerating convolution operations. Table I shows the layer structure of PSPNet (ResNet50). Convolution operations account for most computations in PSPNet. Therefore, we focus on accelerating convolution operations. In the following explanation, we denote the number of channels in the CNN's input and output matrices as IC and OC, respectively, while the convolution filter dimensions are represented as KH and KW for height and width, respectively.

B. im2col

im2col is a technique that accelerates computation by transforming CNN inputs and filters into 2D matrices, effectively converting convolution operations into matrix multiplication. The procedure for im2col consists of the following steps:

1) Convert the input matrix (N, IC, H, W) and filter matrix (OC, IC, KH, KW) into two-dimensional matrices of dimensions $(N \times OH \times OW, IC \times KH \times KW)$ and $(IC \times KH \times KW, OC)$, respectively.
2) Compute the matrix product of the transformed input and filter matrices to obtain an output matrix of dimensions $(N \times OH \times OW, OC)$.
3) Reshape the output matrix $(N \times OH \times OW, OC)$ into a four-dimensional matrix (N, OC, OH, OW) to yield the desired output.

By following these steps, convolution operations are transformed into matrix multiplication, thus, the problem of CNN acceleration is converted into optimizing matrix multiplication performance.

C. Acceleration of Matrix Multiplication on Latest CPU

In this section, a method for accelerating matrix multiplication that utilizes the features of the AMD Ryzen 9 9950X CPU is described with the aim of accelerating the convolution operation using im2col. Table II shows the performance of the Ryzen 9 9950X. The Ryzen 9 9950X is a 16-core, 32-thread CPU that supports AVX2 and AVX-512 as SIMD (Single Instruction, Multiple Data) instruction sets. In addition, each core has two floating-point arithmetic units that support fused multiply-add (FMA) instructions, which perform multiplication and addition simultaneously with a single SIMD instruction. By combining multiple cores and SIMD instructions, it is possible to perform high-density parallel computing.

Algorithm 1 shows a naive implementation of matrix multiplication. Algorithm 1 is the simplest implementation of

Algorithm 1 Naive Matrix Multiplication

Require: $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$
Ensure: $C \in \mathbb{R}^{M \times N}$
for $m \in 0, \ldots, M$ **do**
  for $n \in 0, \ldots, N$ **do**
    for $k \in 0, \ldots, K$ **do**
      $C[m, n] += A[m, k] \times B[k, n]$
    end for
  end for
end for

Algorithm 2 Cache Blocking

Require: $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$
Ensure: $C \in \mathbb{R}^{M \times N}$
for $mc \in 0, \ldots, MC$ **do**
  for $k \in 0, \ldots, KC$ **do**
    load $A_{mc,kc} \leftarrow A[mc, k]$
    for $nc \in 0, \ldots, NC$ **do**
      load $B_{nc,kc} \leftarrow B[k, nc]$
      for $mr \in 0, \ldots, MR$ **do**
        for $nr \in 0, \ldots, NR$ **do**
          $kernel\_matmul(C_{mr,nr}, A_{mr,kc}, B_{kc,nr})$
        end for
      end for
    end for
  end for
end for

matrix multiplication that calculates the product of matrices $A(M \times K)$ and $B(K \times N)$ and yields the output matrix $C(M \times N)$. Using this as a baseline, we will accelerate matrix multiplication.

1) Parallelization with SIMD Instructions: SIMD instructions are instructions that process the same operation on multiple data simultaneously, allowing the CPU to perform more simultaneous operations. The Ryzen 9950X supports the AVX-512 instruction set, and has 32 512-bit wide zmm registers that can process 16 single-precision floating-point numbers (FP32) simultaneously per single register. Furthermore, by using the fused multiply-add (FMA) instruction, it is possible to perform multiplication and addition simultaneously with a single instruction, making it possible to further parallelize matrix multiplication calculations.

2) Utilization of L1, L2, L3 Caches: Because the elements of a matrix are read multiple times in matrix multiplication, the overhead of memory access is large in a simple implementation such as Algorithm 1. Therefore, we reduce the overhead of memory access by implementing cache blocking inspired by the matrix operation library BLIS [21]. In cache blocking, the input and output matrices are first divided into small matrices called blocks, which are then further divided into smaller matrices called panels, and matrix operations are performed. This two-stage partitioning strategy allows the three CPU caches (L1, L2, and L3) to be used efficiently. As shown in Table II, the Ryzen 9950X has large CPU caches compared with other CPUs, so cache blocking can be expected to have a significant effect.

Algorithm 2 shows the implementation of matrix multiplication using cache blocking. In Algorithm 2, the input

Algorithm 3 Matrix Multiplication using SIMD Instructions

Require: A_{mr}, B_{nr},
  A_{reg}(SIMD Register),
  B_{reg}(SIMD Register),
  $C_{reg}[MR]$(SIMD Register)
Ensure: $C_{mr,nr}$
for $mr, nr \in 0, \ldots, MR, NR$ **do**
  $C_{reg}[mr] \leftarrow$ (load_ps) $C_{mr}[mr]$
end for
for $k \in 0, \ldots, K$ **do**
  for $nr \in 0, \ldots, NR$ **do**
    $B_{reg} \leftarrow$ (load_ps) $B_{nr}[nr]$
    for $mr \in 0, \ldots, MR$ **do**
      $A_{reg} \leftarrow$ (set1_ps) $A_{mr}[mr]$
      FMADD$(A_{reg}, B_{reg}, C_{reg})$
    end for
  end for
end for

matrices are $A(M \times K)$ and $B(K \times N)$, and the output matrix is $C(M \times N)$. First, the input matrices A and B are divided into blocks $A_{MC,KC}$ and $B_{KC,NC}$. At this point, by storing the elements of each block in a one-dimensional array, we can maintain the continuity of memory access in the subsequent matrix multiplication calculation. Next, the block matrices $A_{MC,KC}$ and $B_{KC,NC}$ are further divided into panels $A_{MR,KC}$ and $B_{KC,NR}$. Finally, the matrix multiplication using SIMD instructions shown in Algorithm 3 is performed on the panel matrices $A_{MR,KC}$ and $B_{KC,NR}$, and the corresponding output matrix $C_{MR,NR}$ is obtained.

As shown in Figure 1, by using cache blocking, it is possible to place the matrices $B_{KC,NC}$ in the L3 cache, the matrices $A_{MC,KC}$ in the L2 cache, and the matrices $B_{KC,NR}$ in the L1 cache. For the matrix C, since the number of accesses to each element is less than that for the matrices A and B, it is loaded directly from the main memory into the SIMD register. The block and panel division units are set to MC, NC, KC, MR, and NR according to the CPU cache capacity. At this time, by setting the value of NR to a multiple of 16, it is possible to perform matrix multiplication calculations by making the most of the bit width of the zmm register. A multiple of 16 is chosen because the zmm register can handle 16 floating-point numbers (FP32) at a time.

3) Multi-core Parallelization: According to the implementation of cache blocking, the operation of calculating the matrix product for each panel $A_{MR,KC}$ and $B_{KC,NR}$ is independent, so it can be calculated in parallel for each CPU core. As shown in Table II, the Ryzen 9950X is a 16-core, 32-thread CPU, so the matrix product is calculated in parallel using 16 cores.

4) Filter Size: Table I shows the layer structure of PSPNet. There are three types of filter sizes in the PSPNet convolution layer: 1×1, 3×3, and 7×7. For a convolution layer with a filter size of 1×1, the filter matrix $OC \times IC \times 1 \times 1$ is treated as a $OC \times IC$ 2D matrix, which allows matrix multiplication to be applied directly without using im2col. Therefore, in terms of implementation, we apply the matrix product directly to convolution layers with a filter size of

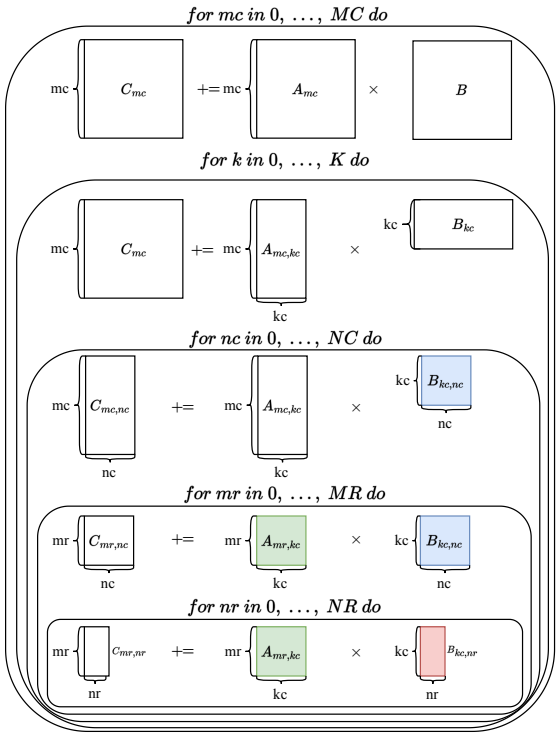

Fig. 1. Overview of matrix multiplication using cache blocking: The red parts are placed in L1 cache, the green parts in L2 cache, and the blue parts in L3 cache. The for loops in the figure correspond to the respective for loops in Algorithm 2.

1×1, and for other filter sizes, we apply im2col to perform the calculation.

D. Fusion of Convolution and Batch Normalization

By pre-computing the learned parameters based on the following Equation 1 and using them as the parameters for a new convolution layer, it is possible to merge the Batch Normalization layer with the Convolution layer.

$$\begin{aligned} Batch\ Norm &= W_{bn}\frac{X - mean}{\sqrt{var + eps}} + bias_{bn} \\ &= \gamma\hat{x} + \beta \\ Fused\ Weight &= W_{conv} \cdot \left(\frac{\gamma}{\sqrt{var + eps}}\right) \\ Fused\ Bias &= \beta_{conv} + \frac{\gamma}{\sqrt{var + eps}}\left(\beta_{bn} - mean\right) \end{aligned} \tag{1}$$

The W_{bn}, $mean$, var, γ, and β in Equation 1 are the parameters of the Batch Normalization layer, and W_{conv} and β_{conv} are the learned parameters of the Convolution layer, respectively. By fusing the Batch Normalization layer of PSPNet using Equation 1, the amount of computation can be reduced.

E. Other Layers in PSPNet

In the previous section, we accelerated the Convolution and Batch Normalization layers of Table I. As shown in Table I, PSPNet also has layers such as ReLU and MaxPooling in addition to the Convolution and Batch Normalization layers. We also parallelized these layers using multi-core parallelization and AVX-512 instructions, and we accelerated the entire PSPNet.

TABLE III
EVALUATION ENVIRONMENT

OS	Ubuntu 24.04 LTS
CPU	AMD Ryzen 9 9950X
RAM	DDR5 5600MT/s 192GB

IV. EVALUATION

This section evaluates the effectiveness of the proposed implementation techniques described in Section III. First, we perform a performance evaluation of matrix multiplication, and then we evaluate the Convolution layer. Finally, we evaluate the effectiveness of the proposed technique in a semantic segmentation task on the PSPNet. The evaluation is conducted using single-precision FLOPS and execution time, with the average value obtained from multiple executions. In addition, the output values are completely identical between the PyTorch implementation and our implementation.

A. Evaluation Setup

The evaluation environment is summarized in Table III. The implementation is performed using the C/C++ language, and the compiler used is gcc v13.3.0 with the optimization options "-O3 -march=native". The OpenMP library is used for multi-core parallelization, and the number of threads in OpenMP is set to the same as the number of physical cores (16). For evaluation, the version of Python is 3.11.9, PyTorch is 2.5.1, and Torchvision is 0.20.1.

B. Theoretical Floating-point Performance

In general, the theoretical value of FLOPS of a CPU is calculated as follows:

$$\begin{aligned} \text{Theoretical FLOPS} = &\text{ CPU Clock} \\ &\times \text{CPU Cores} \\ &\times \text{Processing Elements} \\ &\times \text{Operations per Cycle} \\ &\times \text{Floating Point Operation Units} \end{aligned} \tag{2}$$

Using Equation 2 and the values in Table II, the theoretical performance of the Ryzen 9 9950X is calculated as follows:

$$\begin{aligned} &\text{Theoretical FLOPS of Ryzen 9 9950X} \\ &= 4.3GHz \times 16cores \times \frac{512bit}{32bit} \times 2ops \times 2units \\ &= 4403.2GFLOPS \end{aligned} \tag{3}$$

The clock frequency of the Ryzen 9 9950X when using AVX-512 instructions and all cores is not publicly available, so the base clock of 4.3 GHz is used in Equation 3. The actual theoretical performance is expected to be lower than this value due to thermal throttling and power management.

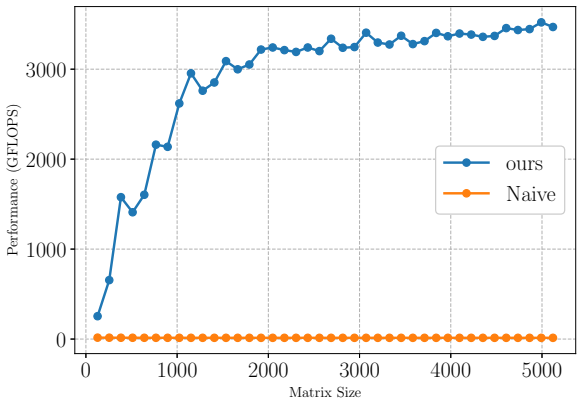

Fig. 2. Evaluation of Matrix Multiplication

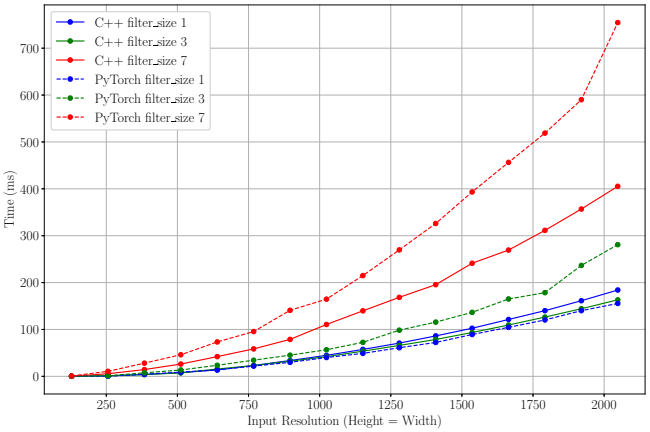

Fig. 3. Evaluation of Convolution Layer

C. Evaluation of Matrix Multiplication

This section evaluates the performance of the matrix multiplication implemented in Section III. For comparison, we use the naive implementation of Algorithm 1 that only applies multi-core parallelization. The size of the matrix to be evaluated is $M = N = K = 128, 256, 384, 512, \ldots, 5120$, with the same width and height for each matrix. In the experiments, as described in Section III-C.2, the values of MC, NC, MR, and NR were set by taking into account the sizes of the L1, L2, and L3 caches, and the values that were the fastest in the experiments were used. Figure 2 shows the performance of matrix multiplication. The results show that the proposed method achieves higher performance than the naive implementation. The highest measured FLOPS value is 3520.64 GFLOPS for $M = N = K = 4992$, which is 79.9% of the theoretical value calculated using Equation 3. Therefore, the proposed method achieves high performance that matches the characteristics of the Ryzen 9950X.

D. Evaluation of Convolution Layer

In this section, we evaluate the performance of the convolutional layers. The convolutional operations to be evaluated have 3 input channels and 64 output channels, and use three filter sizes: 1×1, 3×3, and 7×7. The size of the input image is $1 \times 3 \times H \times W$, where $H = W = 128, 256, 384, 512, \ldots, 2048$. As a comparison, PyTorch's nn.Conv2d is used. Figure 3 shows the performance of the Convolution layer. When the filter size is 1×1, the PyTorch implementation is slightly faster, but when the filter size is 3×3 or 7×7, the proposed implementation shows higher performance than the PyTorch implementation. When the input resolution is high, the difference widens, and the superiority of the proposed implementation is clearly apparent. This is because the parallel performance of the proposed implementation is high, so when the input size or filter size increases and the matrix to be calculated by im2col becomes larger, the effect of parallelization becomes greater.

E. Evaluation of Semantic Segmentation using PSPNet

Finally, we evaluated the performance of the semantic segmentation task using PSPNet. For evaluation, we use weights trained on the Cityscapes Dataset [22] and a dataset created in our research lab for visual navigation, and the input image sizes are $1 \times 3 \times 512 \times 1024$ and $1 \times 3 \times 480 \times 640$, respectively. For comparison, we measure the execution time of the PyTorch implementation. Figure 4 shows the predicted results of PyTorch and C/C++ on the Cityscapes Dataset. As shown in Figure 4, the inference results are the same for the PyTorch and C/C++ implementations, and it has been confirmed that the mIoU evaluation values for both datasets are actually the same. The proposed implementation achieved a performance improvement of $+47.712\%$ on the Cityscapes Dataset and $+39.096\%$ on the Visual Navigation Dataset. As with the evaluation in Figures 2 and 3, the superiority of the proposed implementation becomes more significant as the input resolution increases. This result suggests that the proposed implementation has high parallel computing performance.

TABLE IV
EVALUATION OF PSPNET

Dataset	Cityscapes	Visual Navigation
Input Size	512×1024	480×640
PyTorch	315.612ms	166.622ms
C/C++	**213.667ms**	**119.500ms**
Speedup	**+47.712%**	**+39.096%**

From the above results, the proposed method shows higher performance as the input resolution increases and the size of the matrix handled by im2col increases. When the input resolution is small, the block-divided sub-matrix becomes small, so unless it is divided into an even smaller sub-matrix, the CPU's multi-core performance cannot be fully utilized, resulting in insufficient speedup. Therefore, to achieve even higher speeds, a method will be needed to change the size of the blocks according to the CNN parameters, such as the filter size, similar to the approach used in Intel oneDNN [23].

(a) RGB Image (b) Predicted Label Image (PyTorch) (c) Predicted Label Image (C/C++)

Fig. 4. An example result of semantic segmentation

V. Conclusion

In this study, we proposed a high speed implementation of PSPNet optimized for x86 CPUs having the AVX-512 instructions without any degradation of the inference accuracy. In addition to the SIMD implementation, parallel operation by multi-cores and cache optimization considering L1, L2, and L3 cache memories were applied to speed up the convolution layer that requires huge computational cost at the inference process of PSPNet. To achieve further speed-up, fusion of the convolution layer and the batch normalization layer was proposed. The implemented matrix multiplication reached more than 79% of the theoretical performance of the CPU, and outperformed the PyTorch implementation for 3×3 and 7×7 convolution layers. Furthermore, we succeeded in increasing the speed of PSPNet while keeping the accuracy and achieved a speed increase of 47.712% in the Cityscapes dataset and 39.096% in the Visual Navigation dataset. In the future, we focus on achieving real-time inference speed by quantizing to int8.

References

[1] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 790–799, 1995.

[2] D. Cremers, M. Rousson, and R. Deriche, "A review of statistical approaches to level set segmentation: integrating color, texture, motion and shape," *International Journal of Computer Vision*, vol. 72, no. 2, pp. 195–215, 2007.

[3] Z. Kato and T.-C. Pong, "A markov random field image segmentation model for color textured images," *Image and Vision Computing*, vol. 24, no. 10, pp. 1103–1114, 2006.

[4] X. He, R. Zemel, and M. Carreira-Perpinan, "Multiscale conditional random fields for image labeling," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, vol. 2, 27 Jun.–2 Jul. 2004, pp. II–695–II–702.

[5] Y. Boykov and G. Funka-Lea, "Graph cuts and efficient n-d image segmentation," *International Journal of Computer Vision*, vol. 70, no. 2, pp. 109–131, Nov 2006.

[6] R. J. Jevnisek and S. Avidan, "Co-occurrence filter," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2017, pp. 3816–3824.

[7] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid scene parsing network," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6230–6239.

[8] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, "SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34, 2021, pp. 12 077–12 090.

[9] P. Taghavi, R. Langari, and G. Pandey, "Swinmtl: A shared architecture for simultaneous depth estimation and semantic segmentation from monocular camera images," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2024, pp. 4957–4964.

[10] J. Morioka and R. Miyamoto, "3D Structure Estimation of Room Environment Using Semantic Segmentation and Model Fitting," in *IEEE International Workshop on Metrology for Living Environment*, 2024, pp. 448–453.

[11] M. Wada, Y. Ueda, J. Morioka, M. Adachi, and R. Miyamoto, "Dataset creation for semantic segmentation using colored point clouds considering shadows on traversable area," *Journal of Robotics and Mechatronics*, vol. 35, no. 6, pp. 1406–1418, 2023.

[12] R. Miyamoto, M. Adachi, H. Ishida, T. Watanabe, K. Matsutani, H. Komatsuzaki, S. Sakata, R. Yokota, , and S. Kobayashi, "Visual navigation based on semantic segmentation using only a monocular camera as an external sensor," *Journal of Robotics and Mechatronics*, vol. 32, no. 6, pp. 1137–1153, 2020.

[13] M. Wada, M. Adachi, Y. Ueda, and R. Miyamoto, "Dataset for semantic segmentation generated from 3D scanned data for visual navigation," in *IEEE International Conference on Control, Decision and Information Technologies*, 2023, pp. 1711–1716.

[14] R. Miyamoto, M. Adachi, Y. Nakamura, T. Nakajima, H. Ishida, and S. Kobayashi, "Accuracy Improvement of Semantic Segmentation Using Appropriate Datasets for Robot Navigation," in *IEEE International Conference on Control, Decision and Information Technologies*, 2019, pp. 1610–1615.

[15] M. Adachi, J. Xue, K. Honda, M. Wada, and R. Miyamoto, "Improvement of visual odometry based on robust feature extraction considering semantics," in *IEEE International Conference on Control, Decision and Information Technologies*, 2023, pp. 1705–1710.

[16] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," Oct. 2006.

[17] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[18] V. Ferrari, R. Sousa, M. Pereira, J. a. P. L. De Carvalho, J. N. Amaral, J. Moreira, and G. Araujo, "Advancing direct convolution using convolution slicing optimization and isa extensions," *Transactions on Architecture and Code Optimization*, vol. 20, no. 4, pp. 1–26, Dec. 2023. [Online]. Available: https://doi.org/10.1145/3625004

[19] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4013–4021.

[20] Y.-T. Chen, Y.-F. Ou, and C.-T. Huang, "A winograd-based highly-parallel convolution engine for 8-bit cnn acceleration," in *IEEE International Conference on Artificial Intelligence Circuits and Systems*, 2022, pp. 395–398.

[21] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, pp. 14:1–14:33, June 2015.

[22] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2016, pp. 3213–3223.

[23] oneDNN Contributors, "oneAPI Deep Neural Network Library (oneDNN)," Jan. 2024. [Online]. Available: https://github.com/oneapi-src/oneDNN