

EECE372 Final Project

20190445 허수범

June 11, 2024

1 성능 Bottleneck 분석

연산을 가속화하기 이전, 먼저 각 layer별로 실행 시간이 얼마나 소요되었는지를 확인하였습니다. 이는 Figure 3a와 Figure 4a에서 확인할 수 있습니다. 확인 결과, 실행 시간의 대부분이 convolutional layer에서 소요되었고, 그 다음으로 많이 소요된 곳은 zero padding layer였습니다.

효과적인 가속 방법을 찾기 위해 먼저 convolutional layer의 kernel과 input의 해당되는 3×3 영역의 내적에 다양한 가속 방법들을 적용해 보았습니다. 확인해 본 방법들은 OpenMP를 활용한 multithreading, NEON을 활용한 병렬 연산, inline assembly로 변환되었습니다. 이 중 OpenMP와 NEON intrinsic은 예상만큼 큰 실행 시간의 단축을 보여주지 않고 실행 시간이 매우 불규칙적이었으며, NEON arm assembly로 구현하는 것은 구현의 난이도가 매우 높았습니다. 반면 parallel programming을 사용하지 않고 단순히 assembly로 구현하는 것은 실행 시간을 일관적으로 확연히 단축해 주었습니다.

또한, 다양한 가속 방법에 대한 비교 분석을 해본 결과, nested for loop에서 중첩된 for loop의 갯수를 줄이는 것이 실행 시간을 크게 단축시켜준다는 것도 알게 되었습니다. 같은 내용의 코드더라도, for loop을 사용하지 않고 길게 늘여 쓰는 것이 코드의 실행 시간이 빠르고, 이는 assembly에서도 마찬가지임을 알게 되었습니다. Assembly로 구현을 하더라도, branch operation의 갯수가 많아지면, 실행 시간이 유의미하게 늘어났습니다.

따라서 본 final project를 수행하며 집중한 부분은 nested for loop의 깊이를 줄이고, 최대한 많은 부분을 inline assembly로 구현하는 것이었습니다.

2 Convolutional Layer Optimization

Section 1의 분석을 토대로, nested for loop의 깊이를 최대한 줄이며, 최대한 모든 부분을 inline assembly로 구현하려 노력했습니다. 시작은 Section 1에 설명했다시피, kernel과 input의 내적을 수행하는 부분을 assembly로 번역하는 것이었습니다. 이후에는, input의 channel에 대한 for loop을 수행하는 것 역시 assembly로 구현하였고, 최종적으로는 ‘void Conv2d()’ 함수 전체를 inline assembly로 구현해 보았습니다. 하지만 놀랍게도 코드의 실행 시간은 함수 전체를 assembly로 번역한 것이 아닌, ‘feature_out’의 channel, height, width를 iterate하는 for loop 세 개를 남겨놓는 것이 가장 실행 시간이 빨랐습니다. 이는 for loop을 assembly로 구현하는 비중이 늘어날수록 branch operation의 갯수가 많아져 어느 수준 이상부터는 컴파일러의 효율이 더 좋아지기 때문인 것으로 생각됩니다.

또한, 단순히 assembly로 번역하는 것 뿐만이 아니라, memory에의 접근 횟수를 최대한 줄이고, inline assembly의 input 변수에 접근하는 횟수를 최대한 줄여 register만으로 최대한 많은 연산을 진행할 수 있도록 해주었습니다. 또한, channel(바깥쪽), height, width(안쪽)순으로 loop를 쌓아 메모리를 최대한 연속적으로 접근할 수 있도록 하였습니다. 이 principle은 다른 layer들을 가속할 때도 적용하였습니다.

3 Zero Padding Optimization

Zero padding 역시 Section 1의 분석을 토대로 최대한 inline assembly로 구현하려 노력했고, Section 2와 같이 nested for loop을 하나씩 줄여나가며 실행 시간을 측정하였습니다. 그 결과, channel에 대한 for loop 하나만을 c언어로 남겨놓은 것이 가장 실행 시간이 빨라 해당 방식을 채택하였습니다.

4 Other Layers

나머지 layer들을 ‘Other Layers’라는 하나의 묶음으로 취급한 까닭은, Figure 3a와 Figure 4a에서와 같이 layer별 실행 시간을 측정했을 때 convolutional layer와 zero padding layer를 제외한 layer들의 실행 시간이 극히 작은 비중을 차지하였기 때문입니다. 하지만, convolutional layer와 zero padding layer를 모두 assembly로 변환하고 나니, 전체 실행 시간이 $1000\mu\text{s}$ 대로 줄어들어, ‘Other Layers’들의 실행 시간이 무시할 수 없는 수준이 되었습니다. 따라서 ‘Other Layers’를 역시 inline assembly로 번역해 주었습니다. 이 때 loop의 갯수를 미리 알 수 있는 ‘Get_pred()’의 경우, branch operation을 사용하지 않고, 모든 iteration을 풀어 쓰는 방법을 택했습니다. 이는 이전 실험에서 branch operation의 갯수가 속도에 영향을 미친다는 것을 확인하였기 때문입니다. 다른 layer들의 경우에는 단순히 c언어를 inline assembly로 번역해 주는 방식으로 가속하였습니다.

5 Results

본 project의 다양한 실험 결과는 Figures 1, 2, 3, 4에서 확인할 수 있습니다. 먼저 직접 활용한 이미지에 대한 실험 결과는 Figures 1, 2에서 확인할 수 있는데, 모두 정상적으로 사진이 촬영되었으며, softmax output, activation map의 출력, 7-segment display의 출력 모두 정상적으로 실행되었다는 것을 볼 수 있습니다. 어렵게도 prediction result는 정답과는 다른 결과가 나왔지만, 이는 학습된 dataset과 captured image 사이의 간극에 의한 것이라고 생각됩니다. Example mode의 실행 결과는 Figures 3, 4에서 확인할 수 있는데, 가속되지 않은 코드와 가속된 코드 모두 example result와 동일한 결과가 출력되었고, Activation map 역시 출제 파일의 예시와 동일하게 출력된 것을 확인할 수 있습니다.

실행 시간의 경우, 가속되지 않은 코드는 $5000\mu\text{s}$ 이상이 소요되는 반면, 가속된 코드의 경우 평균적으로 약 $600\mu\text{s}$ 의 시간이 소요되는 것을 확인할 수 있습니다. Inline assembly의 활용과 for loop의 해체를 통해 실행 시간을 대략 8배 이상 단축시키는 데에 성공하였습니다.

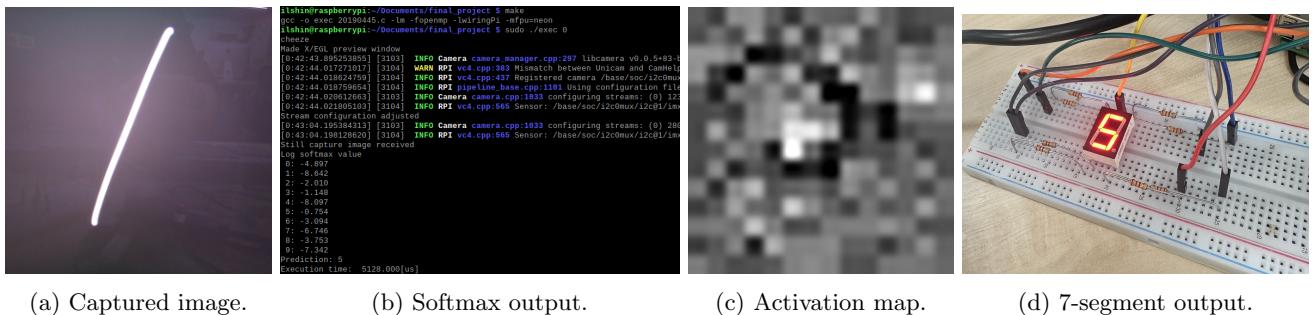


Figure 1: 직접 촬영한 이미지와 inference 결과(가속되지 않은 코드).

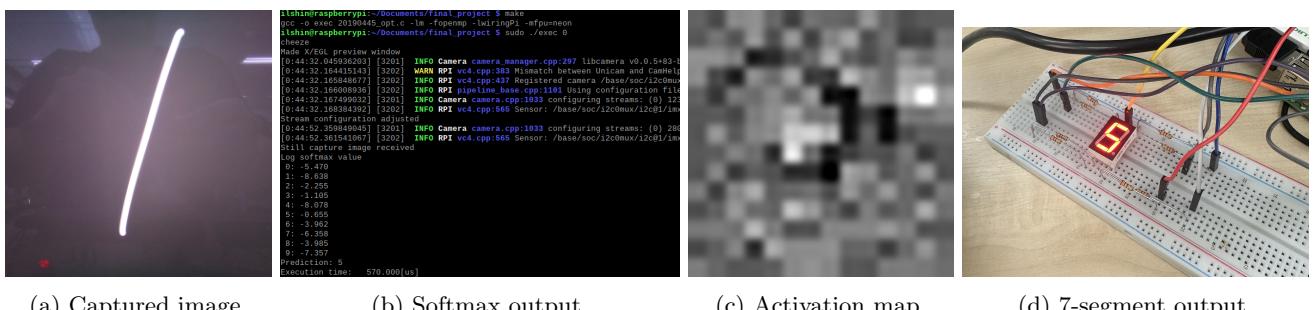
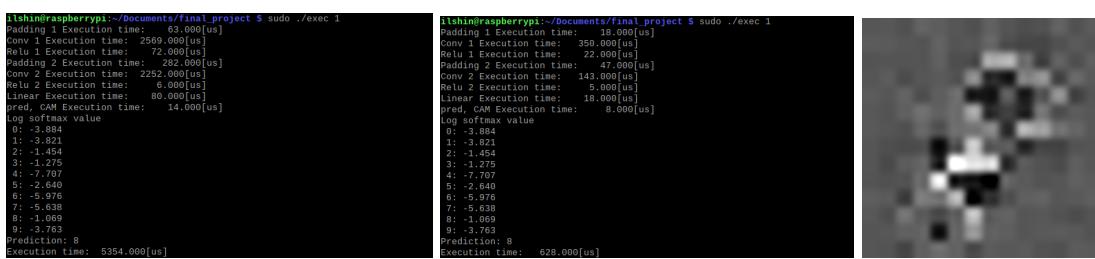
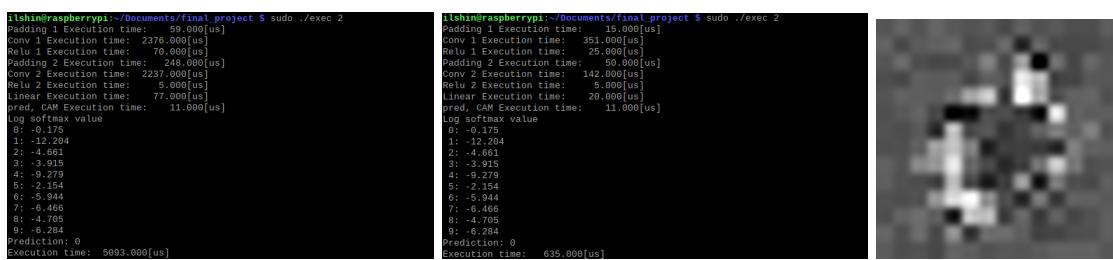


Figure 2: 직접 촬영한 이미지와 inference 결과(가속된 코드).



(a) Softmax output(가속되지 않은 코드). (b) Softmax output(가속된 코드). (c) Activation map.

Figure 3: Example mode results(example1.bmp).



(a) Softmax output(가속되지 않은 코드). (b) Softmax output(가속된 코드). (c) Activation map.

Figure 4: Example mode results(example2.bmp).