모바일 기기 기반의 Stable Diffusion 최적화

(The Optimization of Stable Diffusion based on the Mobile)

지도교수 : 유승주

이 논문을 공학학사 학위 논문으로 제출함.

2023년 12월 20일

서울대학교 공과대학

컴 퓨 터 공 학 부

구 현 우

2024 년 2 월

모바일 기기 기반의 Stable Diffusion 최적화

(The Optimization of Stable Diffusion based on the Mobile)

지도교수 : 유승주

이 논문을 공학학사 학위 논문으로 제출함.

2023년 12월 20일

서울대학교 공과대학

컴 퓨 터 공 학 부

구 현 우

2024 년 2 월

국문초록

모바일 기기에 들어가는 System on Chip(SoC)의 성능은 날이 감에 따라 발전하고 있다. On-Device AI 기술은 제한적인 환경에 대한 단점이 존재하지만 보안 문제와 비용 문제에 대한 좋은 해결책을 제시할 수 있다. 본 연구에서는 Stable Diffusion 모델을 삼성의 Galaxy S21 Exynos 2100 기기 위에서 최적화를 하였다. 오픈 소스인 Stable Diffusion v2.1의 Pytorch 구현체를 기반으로 Android 앱에서 Android NDK와 C/C++ 및 OpenCL을 활용하여 앱을 작성하였다. 최적화 기법에 대한 성능 향상을 통해 비교군인 Pytorch 모델과 성능 비교를 하였다. On-Device AI의 실생활 적용의 가능성을 확인하고 추가적인 최적화 방안에 대한 제언을 하였다.

주요어 : On-Device AI, OpenCL, Stable Diffusion, 모델 최적화

제 1 장 서론

제 1 절 On-Device AI

모바일 기기에 들어가는 System on Chip(SoC)의 성능은 날이 감에 따라 발전하고 있다. 현대인은 스마트폰을 기본으로 태블릿 PC, 스마트 워치 등 다양한 모바일 기기를 소유하고 이는 고성능 Edge-Device의 대중화로 이어지고 있다. 기술의 발전을 주도하고 있는 AI 분야는 Large Language Model(LLM)의 발전과 하나의 모델이 여러 문제에서 준수한 성능을 보여주는 파운데이션 모델의 발전은 AI 기술이 실생활에 적용되는 초석을 마련하였다. 현재 LLM분야는 파라미터 수가 GPT-4의 경우 약 1 조에 달하는 크기로 FP32 기준 약 4 Terabytes 크기로 엄청 큰 크기의 모델로 성능을 향상시켰다.[1] 모델의 크기를 키워 성능을 높이는 발전에는 많은 비용과 시간이 소모되며 이를 실생활에 적용시키기 위해서 모델의 성능을 최적화하여 적은 비용과 시간으로 모델을 구동할 수 있어야 할 것이다. 본 연구에서 이러한 효율적인 모델의 구동과 한정된 자원의 환경 위에서 최적화하는 실험을 통해 현재 모바일 기기의 성능과 최적화 방식이 달성할 수 있는 수준을 알아보고자 한다.

On-Device AI 기술은 클라우드 기반 AI 보다 제한적인 환경에 대한 단점이 존재하지만 현재 문제점으로 꼽히는 보안 문제와 비용 문제에 대한 좋은 해결책을 제시할 수 있다. 클라우드 기반 AI는 현재 ChatGPT[2], Midjourney[3]와 같은 서비스에서 주로 AI 모델을 배포하는 방식이다. 이는 크기가 큰 모델을 사용자들에게 부담 없이 제공하기에 좋은 방식이지만 이를 운영하는 입장에서는 비용적인 부담이 크게 다가올 수 있다. 사용자들이 제공한 정보를 통해 추가적인 학습을 하는 경우도 있어 학습 데이터에 대한 충분한 익명화가 이루어 지지 않는다면 보안 문제로 이어질 수 있다. 이러한 면에서 On-Device AI는 각 edge device에서 모델의 추론이 이뤄지기 때문에 개인의 컴퓨팅 자원을 최대한 활용할 수 있으면서 개인 정보 유출의 위험성이 없고 네트워크 환경이 온전치 않은 경우에도 잘 동작할

수 있다.[4] On-Device AI의 장점들은 한정된 자원에서 구동되어야 하는 단점을 극복하기만 한다면 실생활에 충분히 적용되기 좋은 기술이다.

제 2 절 OpenCL

AI 분야의 새로운 기술들은 주로 Python 환경에서 NVIDA 의 CUDA 프레임워크 위에서 구동된다. CUDA 는 GPU 성능을 충분히 활용할 수 있게 해주지만 이는 NVIDIA GPU 에 국한된다. 모바일 기기 시장은 다양한 제조사의 하드웨어가 있으며 다양한 제조사가 지원하는 프레임워크로 OpenCL 이 사용된다.[5] OpenCL 은 다양한 제조사를 지원하는 만큼 CUDA 와 달리 최고의 성능으로 최적화되어 있지는 않지만 하나의 코드가 여러 기기에서 쓰일 수 있는 범용성이 장점으로 존재한다. 최근 모바일 기기에서는 GPU 이외에도 NPU 나 TPU 와 같이 모델 추론을 위한 하드웨어도 탑재되기 때문에 프레임워크의 범용성은 필수 불가결한 요소이다. 본 연구에서는 OpenCL 기반의 구현을 통해 모바일 기기의 GPU 에서 연산을 하도록 했으며 이러한 연구가 다른 모바일 기기까지 확장될 가능성을 남겨두었다.

제 3 절 Stable Diffusion

Stable Diffusion 은 이미지 생성형 모델 중 작은 크기로 좋은 성능을 내는 모델이다. 기존 Generative Adversarial Network(GAN) 혹은 Variational Autoencoder(VAE)과 같은 모델이 이미지 생성 모델을 발전을 이끌었다면 최근에는 Diffusion 기반 이미지 생성 모델이 주목을 받고 있다. 원본 이미지에 시간 순서에 따른 노이즈를 추가하는 과정을 Diffusion 이라 불린다. Diffusion 기반 이미지 생성 모델은 Diffusion 과정을 역순으로 적용해 노이즈에서 원본 이미지로 추론을 통해 이미지를 생성하는 기술이다. Diffusion 기반 모델 중에서도 Stable Diffusion 이 현재 작은 크기로 좋은 성능을 내는 모델로 각광받고 있다. Stable

Diffusion 은 기존 이미지 생성 모델과 달리 이미지를 바로 생성하는 것이 아니라 latent space 라는 차원 공간에서 추론을 통해 더 적은 연산으로 이미지를 추론하여 Decode 과정을 통해 latent space 의 값을 RGB 포맷의 이미지로 생성한다.[6] 텍스트나 다른 이미지와 같은 값을 인코딩하여 조건으로 사용할 수 있어 다양한 방식으로 활용될 수 있다. 작은 크기로 다양한 활용도를 가진 Stable Diffusion 은 이미지 생성 모델에서 실생활 적용의 실마리를 제공하여 현재 Midjourney[3] 와 같은 서비스에서 활용하고 있다.

선행 연구에서는 Galaxy S23 Snapdragon 8 Gen 2 기기를 기반으로 Stable Diffusion 모델을 양자화 없이 연산의 최적화만으로 추론 시간을 줄이거나[7] 양자화를 하여 최적화를 하거나[8] Qualcomm 에서 공개한 연구의 경우 자체 모델 최적화 툴을 사용해 모바일에 구동하는[9] 등 On-Device 분야에서 도전적인 과제로써 활용되고 있다. 본 연구에서는 선행연구들과 달리 현존하는 최고의 성능의 기기가 아닌 더 제한적인 환경에서도 해당 모델의 최적화 기법들이 잘 동작하는 알아보고 실생활 적용에 대한 가능성을 확인하려 한다.

제 2 장 환경 및 구현

제 1 절 구동 환경

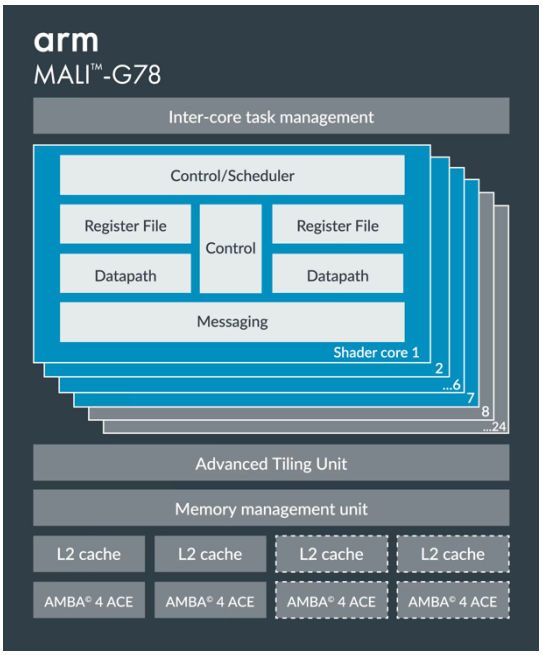

Figure 1. Mali-G78 구조 [10]

Stable Diffusion 모델의 추론을 구동할 모바일 기기는 삼성의 Galaxy S21 Exynos 2100 System on Chip(SoC) 이다.[11] Exynos 2100 칩의 CPU는 ARMv8-A64 64-bit 기반의 2.9GHz Cortex®-X1 싱글코어와 2.8GHz Cortex®-A78 트리플코어 및 2.2GHz Cortex®-A55 쿼드코어로 구성되어 있다. 본 연구에서 비교군으로 구현의 기반이 되는 Stability AI 에서 공개한 Stable Diffusion v2.1 의 구현체를 PyTorch 라이브러리에서 제공하는 JIT 컴파일러를 통해 3 가지 추론 모델들을 컴파일하여 Android OS 위에서 CPU 를 통해 구동하여 기본 성능을 측정하였다. 메인 메모리는 8GB 로 되어 있으며 약 1 Gigabytes 정도만 메모리에 올려서 연산에 활용할 수 있는 환경적 제약이 있다. Exynos 2100 칩의 GPU 는 ARM 사의 Mali-G78 MP14 모델을 사용한다. Mali-G78 MP14 GPU는 2 세대 Valhall 구조 기반으로

Fused Multiply-Add(FMA) 유닛을 탑재해 행렬곱을 효율적으로 할 수 있다. OpenCL 2.0 버전까지 지원하며 최대 24 개의 쉐이더 코어를 지원하고 MP14 는 총 14 개의 쉐이더 코어를 가지고 각 코어는 64 개의 쉐이더를 지원한다.[10] Microsoft 사에서 공개한 모바일 기기의 GPU 프로파일링 툴인 ArchProbe[12] 를 통해 알아낸 정보에 따르면 Mali-G78 MP14 는 1 MegaBytes L2 cache 를 가지고 64 bytes 크기의 cache line 을 가지고 있었다. 약 5.6 Gigabytes 크기의 global memory 를 가지고 있으며 구조적으로 shared memory 는 가지고 있지 않지만 global memory 를 통해 shared memory 를 사용할 수 있도록 되어 있었다. FP32 에 대한 연산 속도는 약 768 Gflops 이고 FP16 에 대한 연산 속도는 약 1532 Gflops 인 것으로 측정되었다. Exynos 2100 칩에는 트리플코어 NPU 및 DSP 를 탑재한 AI 엔진이 존재하지만 이는 OpenCL 을 지원하지 않고 Android NDK 에서 제공하는 NNAPI 를 통해 활용할 수 있다. 모바일의 AI 추론 성능을 비교하는 AI-Benchmark 를 통해 GPU 와 NPU 성능을 비교한 결과 U-Net FP32 모델 기준 NPU 는 605 millisecond(ms)가 소요되었고 GPU 는 204 ms 가 소요되어 더 성능이 좋은 GPU 를 통해 Stable Diffusion 모델을 구동하기 위해 OpenCL 프레임워크를 선택하게 되었다.

Model	SoC	FP16 NNAPI 1.3	FP16 Accuracy	FP16 Parallel	FP16 NLP	AI Score
S23	Snapdragon 8 Gen 2	241	92.3	52.4	77.5	2503
S21	Exynos 2100	60.8	92.9	1.7	6.7	255

표 1. S23 vs S21 성능 비교 [13]

본 연구에서 채택한 Galaxy S21 Exynos 2100 기기는 선행 연구[7,8,9]와 달리 일반적인 성능을 가진 기기 환경에서 최적화를 진행해 현실적인 on-device AI 추론에 대한 가능성을 탐구하였다. 선행 연구에서 채택한 Galaxy S23 Snapdragon 8 Gen 2 기기는 본 연구에서 채택한 Galaxy S21 Exynos 2100 기기보다 성능이 좋은 기기이다. AI-Benchmark 에서 제공하는 성능 비교 표를 살펴 보면 종합적인 AI Score 는 S23 Snapdragon 8 Gen 2 기기가 2503 점으로 S21 Exynos 2100 기기의 255 점보다 약 9.8 배 좋은 평가를 받았다.[13] U-Net[21] FP16 추론 시간

기준 S23 Snapdragon 8 Gen 2 는 23 ms 가 소요된 것에 비해 S21 Exynos 2100 기기는 108 ms 가 소요되어 약 4.7 배 정도 추론 속도에 있어 좋은 성능을 가진것을 알 수 있다.[14] 앞선 비교에서 알 수 있듯이 본 연구는 도전적인 환경에서 Stable Diffusion 을 구동하여 on-Device AI 에 대한 실질적 적용에 대한 탐구를 하였다.

제 2 절 Stable Diffusion

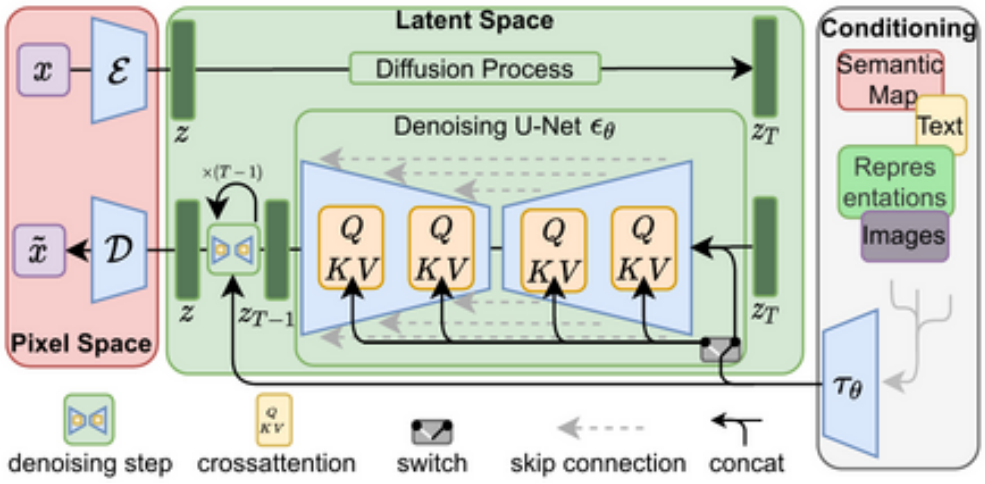

Figure 2. Stable Diffusion 구조[6]

본 연구에서 최적화 대상으로 채택한 Stable Diffusion 모델은 Stability AI 에서 공개한 Stable Diffusion v2.1 의 Pytorch 구현체에 기반해 있다.[15] Pytorch 버전의 Stable Diffusion 은 pytorch 외에도 numpy 및 openclip 등 외부 라이브러리를 통해 구현해 있어 미리 제공된 API 를 통해 추상화가 잘되어 구현되어 있었다. 다양한 설정값에도 잘 동작할 수 있도록 코드가 잘 구현되어 있었지만 이를 OpenCL 로 옮기기에는 불필요한 부분이 많았다. 최소한의 추론만 가능 하도록 텍스트 기반 조건을 통해 이미지를 생성하는 추론 부분만 구현하였다.

Stable Diffusion 의 구현부를 크게 나누면 5 가지로 Tokenizer, Text Encoder, DDIM Sampler, U-Net, Decoder 가 있다. 먼저 자연어 기반 텍스트를 숫자로 토큰화 해주는 Tokenizer 와

Tokenizer 를 통해 나온 토큰을 이미지 생성에서 활용할 수 있도록 인코딩하는 Text Encoder 가 있다. DDIM Sampler 는 시간 순서에 맞게 적합한 조건값과 시간값을 넣어 노이즈부터 생성된 이미지까지 U-Net 을 반복하여 추론하는 역할을 한다. U-Net 은 Stable Diffusion 에서 노이즈에서 각 시간 순서에 맞게 노이즈가 추가되기 이전의 원 이미지를 추론하는 모델로 핵심적인 역할을 하며 차지하는 비중도 크다. 마지막으로 생성된 이미지는 Latent space 에 있는 상태로 이를 다시 RGB 포맷의 이미지로 바꿔주는 역할은 Decoder 가 한다. 이렇게 5 가지 큰 구성요소들이 모여 하나의 Stable Diffusion 을 이룬다. 본 연구에서 사용된 Tokenizer 와 Text Encoder 는 OpenAI 에서 공개한 OpenCLIP[16] 라이브러리를 활용하여 이 두 구성요소는 OpenCLIP 코드를 따라 구현하였다.

Stable Diffusion 의 학습된 weight 와 bias 파일은 Stability AI 에서 Hugging Face 에 공개한 512x512 크기의 이미지를 생성하는 FP32 모델이 학습된 safetensor 파일을 기반으로 구현을 하였다.[17] safetensor 파일에 Stable Diffusion 구현체에 쓰이는 다양한 모델의 weight 와 bias 가 담겨있고 약 5.21 Gigabytes 로 파일의 크기도 크기때문에 이를 OpenCL 구현체에 직접 활용할 수 없어 각 구성요소에 쓰인 weight 와 bias 값을 하나씩 numpy 로 npy 파일로 추출하여 모바일 기기의 저장장소로 옮겨 구현에 사용하였다. 또한 비교군을 위해 Pytorch 에서 제공하는 JIT 컴파일러로 Text Encoder, U-Net, Decoder 를 각각 컴파일하여 Android 의 Pytorch 라이브러리를 통해 구동하였다. Pytorch 모델의 경우 Text Encoder 와 Decoder 는 약 각각 1.42 Gigabytes 와 335 Megabytes 로 메모리에 한번에 올려서 추론할 수 있었지만 U-Net 의 경우 3.48 Gigabytes 로 한번에 메모리에 올렸을 시 메모리 부족으로 앱이 OS 에 의해 정리되어 4 부분으로 나눠 각 부분씩 메모리에 올려가며 나눠서 추론시간을 측정하였다.

제 3 절 OpenCL 구현

앞서 언급한 Pytorch 구현체를 기반으로 Android OS 상 앱안에서 Stable Diffusion 을 구동하기 위해 Android NDK 와 C/C++ 을 활용한 앱을 작성하였다. OpenCL 의 java 기반 라이브러리인 jocl 도 고려를 하였지만 low-level 에서 최적화와 구현한 코드의 범용성을 고려했을 때 C/C++ 기반 OpenCL 을 사용하여 구현하였다. OpenCL 은 ARM Mali-G78 MP14 GPU 에서 자체적으로 지원하지만 Android NDK 에서 기본으로 제공하고 있지 않기 때문에 모바일 기기에 탑재된 OpenCL 의 shared library 를 앱에 넣어 Android NDK 에서 OpenCL 를 사용할 수 있도록 설정하였다. OpenCL 은 GPU 에서 지원하는 버전인 2.0 버전을 사용하였으며 OpenCL kernel 파일은 앱내 assets 에 넣어 Android 의 Asset Manager 를 통해 읽어오도록 하였고 학습된 모델의 weight 와 bias 파일들은 앱의 media 경로에 저장하여 외부 디스크에서 읽을 수 있도록 하였다.

본 연구에서 Android NDK 로 구현한 Stable Diffusion 의 5 가지 구성요소(Tokenizer, Text Encoder, DDIM Sampler, U-Net, Decoder)로 이뤄져있다. 이 중 Tokenizer 와 DDIM Sampler 는 GPU 연산이 불필요한 구성요소로 C++ 기반으로 CPU 상 동작하도록 구현하였다. 다른 3 가지 구성요소(Text Encoder, U-Net, Decoder)는 각각 미리 학습된 weight 와 bias 를 가지고 있으며 OpenCL 기반으로 GPU 연산을 통해 동작하도록 구현하였다.

Text Encoder 는 23 개의 Residual Attention Block 과 1 개의 Layer Normalization 으로 구성되어 있다. Residual Attention Block 은 Normalization 과 Linear 그리고 Multi Head Attention 으로 구성되어 있다. Multi Head Attention 은 또 Linear 과 여러 vector 연산으로 구성되어 있다. 이를 정리하여 구현한 OpenCL kernel 코드들은 살펴보면 행렬의 차원 순서를 바꾸는 permute, batch 단위로 행렬곱하는 batch matmul, Softmax 활성화 함수를 구현한 softmax 가 있다. input 으로 (77) 크기의 [0, 49408) 범위인 정수로 구성된 토큰이 들어와 각 토큰은 (1024) 크기의 값으로 바뀌어 (77, 1024) 크기로 연산에 사용되어 같은 크기의 output 으로 인코딩된다.

U-Net 은 Linear, 2D Convolution, Res Block, Spatial Transformer, Up Sample, Group Normalization 으로 구성되어 있으며 주로 2D Convolution 과 Linear 로 구성되어 있다. U-Net 은 Stable Diffusion 모델 중 상당 부분을 차지하여 모델의 크기도 크고 크기가 큰 행렬들의 연산이 많아 복잡하면서 추론 시간이 가장 길다. input 으로 noise 가 섞인 (4, 64, 64) 크기의 행렬과 시간 순서와 Text Encoder 의 결과값이 들어온다. U-Net 은 latent space 에서 denoise 하는 과정을 거쳐 output 역시 (4, 64, 64) 크기로 도출되어 다음 U-Net 의 추론에 사용된다.

Decoder 는 2D Convolution, Res Block, Attention Block, Up Sample, Group Normalization 으로 구성되어 있으며 2D Convolution 이 연산의 대부분을 이루는 구조를 가지고 있다. Input 으로 U-Net 에서 denoising 한 latent space 의 (4, 64, 64) 크기 행렬이 들어오고 이를 Up Sampling 하면서 (3, 512, 512) 크기의 RGB 포맷의 이미지로 만들어주는 역할을 한다.

각 모델은 재활용되는 Layer 도 존재하여 정리를 하면 OpenCL kernel 로 구현한 연산은 2D Convolution, matrix multiplication, mean, variance, normalization, Softmax, GELU, SiLU, up sampling, element-wise add, multiply, 행렬의 차원 순서를 바꾸는 permute 를 직접 구현하였다.

첫 구현은 Naive 한 구현으로 최적화를 구현하지 않고 논리적으로 모델이 동작할 수 있도록 구현을 하였다. Naive 한 구현 후에 ARM Mali-G78 MP14 GPU 에 최적화 하는 과정을 추가적으로 거쳐 개선을 하였다.

제 3 장 최적화

제 1 절 Reduction

GPU 의 Parallelism 을 활용하기 위해 Mean, Variance, Softmax 의 kernel 구현에 reduction 을 적용하였다. Reduction 은 동일한 work group 에서 실행되는 shader 는 shared memory 를 공유할 수 있고 Single Instruction Multiple Data(SIMD)으로 동작하는 GPU 구조를 활용하여 $O(n)$ 시간의 연산을 $O(log_2(n))$ 시간으로 줄일 수 있다. Normalization 에 사용되는 mean 과 variance 를 구하는 kernel 에서 reduction 을 적용했고 Softmax 를 적용하는 kernel 에도 sum 이 활용되어 reduction 을 적용할 수 있었다. 본 연구에서 Naive 한 구현에서 reduction 을 활용하여 구현했기 때문에 따로 reduction 적용 전후의 소요 시간에 대한 비교는 하지 않았다.

제 2 절 Local memory & Register Utilization

행렬곱의 연산에서 tiling 을 적용하여 work group 이 하나의 tile 에 해당하는 결과에 필요한 연산을 tile 단위로 local memory 에 가져와 처리하여 global memory 에 대한 접근을 줄일 수 있었다. shader 하나 당 여러개의 결과값을 계산하여 register 에 필요한 값을 넣어 재사용하여 local memory 에 대한 접근을 cache 로 옮겨 최적화 할 수 있었다. Naive 한 행렬곱 연산에서는 결과값의 한 요소를 global memory 에 접근하며 연산하여 비효율적이었다. 이를 메모리 계층상 더 빠른 메모리에 캐싱하는 방식으로 연산 속도를 올릴 수 있었다. Stable Diffusion 에서 2D Convolution 과 함께 연산의 대부분을 구성하는 Linear 에서 행렬곱이

쓰이며 Attention 을 구하거나 Up Sample 하는 연산에도 행렬곱이 쓰여 이를 개선하는 것만으로도 연산 시간을 줄이는 것에 기여할 수 있었다.

제 3 절 im2win 2D Convolution

2D Convolution 을 메모리를 더 사용하면서 연산 시간을 효과적으로 줄일 수 있는 im2win[18] 방식을 통해 최적화 하였다. Naive 한 2D Convolution 의 구현은 결과값의 한 요소를 계산하기 위해 입력값과 filter 의 모든 channel 을 접근하여 연산을 하는 방식으로 메모리 접근에 비효율적인 방식이었다. 이를 개선하기 위해 주로 쓰이는 im2col[19] 방식은 입력값인 행렬을 결과값에 필요한 연산을 행렬곱으로 처리하기 위해 필요한 입력값 3 차원 행렬(image)의 요소들을 복사하여 최종적으로 2 차원 행렬(column)으로 만들어 효율적인 메모리 접근을 한다. im2col 방식은 Naive 한 방식보다 메모리는 더 많이 사용하지만 메모리 접근의 효율성을 높이고 행렬곱의 최적화 기법을 적용하여 연산을 최적화할 수 있는 장점이 있다. 하지만 모바일 환경에서 제한적인 메모리 상황에서 입력값 행렬의 차원의 크기가 커지면 메모리 소모가 너무 커져 사용하기 부담이 되었다. 본 연구에서도 U-Net 모델은 latent space 에서 이미지에 비해 작은 차원의 크기로 사용하는 것에 문제는 없었지만 Decoder 모델은 latent space 에 있는 값을 이미지로 키우는 것이기 때문에 이 과정에서 너무 많은 메모리가 필요해 연산도중 메모리 부족으로 되지 않는 상황을 겪었다.

Im2win 방식은 Im2col 방식이 반복된 메모리를 많이 소모하는 것에 착안하여 메모리의 재사용율을 높이고자 고안된 방식으로 상황에 따라 im2col 방식 대비 ⅓ 크기의 메모리만으로 더 빠른 속도로 연산을 할 수 있었다. Im2win 방식은 기존 im2co 과 달리 filter 의 window 에 해당하는 값에 대해서 column 값을 row 로 재배치 하여 필터가 다음 값을 계산하여 움직이더라도 값을 재사용할 수 있도록 한다. im2win 방식은 im2col 보다 메모리는 적게 사용하면서 유사하거나 때론 더 좋은 성능을 보이기도 한다. 하지만 im2win

방식으로 변환된 입력값 행렬은 행렬곱을 그대로 적용할 수는 없는 단점을 가진다. 메모리가 제한적인 모바일 환경에서 im2col 방식을 대체할 수 있는 좋은 방식으로 본 연구에서 2D Convolution 을 최적화 하는 방식으로 채택하여 성능 개선에 도움이 되었다.

제 4 장 결론 및 정리

제 1 절 한정적인 메모리 문제 및 해결

Galaxy S21 Exynos 2100 환경은 8 Gigabytes 의 메인 메모리와 GPU 에 5.6 Gigabytes global memory 가 존재한다. Android OS 의 Low Memory Killer Demon(lmkd)[20] 프로세스는 메모리 상태를 모니터링 하다가 메모리 부족 상태가 되면 최소한의 프로세스를 죽여 메모리를 확보한다. lmkd 프로세스는 앱의 과다한 메모리 사용을 방지하여 Stable Diffusion 의 U-Net 모델의 3.48 Gigabytes 크기의 weight 와 bias 를 한번에 메모리에 올릴 수 없게 했다. U-Net 모델을 구조상 input block 에 해당하는 층이 (4, 64, 64) 크기의 값을 channel 을 늘리고 width 와 height 는 줄여 (2560, 8, 8) 크기의 값으로 점차적으로 만들고 output block 에 해당하는 층은 input block 의 층의 수와 같으며 반대의 역할로 이전 층의 결과값과 input block 의 각 층의 결과값을 residual mapping 을 통해 전달받아 다음 층으로 결과값을 도출한다. 이 처럼 output block 은 대칭적으로 존재하는 input block 의 층의 결과값이 필요했고 이는 모델을 나누는 것에 어려움이 되었다.

비교군을 위한 Pytorch 모델의 경우 JIT 컴파일러로 컴파일된 모델이라 tensor 형태의 입력값을 받을 수 있었다. 이를 해결하기 위해 input block 의 각 층의 결과값들을 1 차원 tensor 로 stack 하여 각 output block 에서 필요한 만큼 pop 하는 방식으로 모델을 나눠 값을 처리할 수 있었다.

OpenCL 로 구현한 U-Net 의 경우는 각 층의 연산에 필요한 weight 와 bias 값만 메모리에 올려두고 해당 층의 연산이 마친후 메모리를 해제하는 방식으로 I/O 의 overhead 에 대한 비용이 들지만 한번의 과정으로 연산을 할 수 있도록 하였다.

제 2 절 성능 비교

Target	Text Encoder		U-Net		Decoder	
	Inference w/ IO	Inference	Inference w/ IO	Inference	Inference w/ IO	Inference
Pytorch(CPU)	8.9s	2.6s	36.2s	13.4s	14.6s	13.4s
Naïve(GPU)	-	17.4s	17m 35s	-	-	29m 4s
Optimized(GPU)	-	8.4s	7m 6.6s	-	3m 6.7s	-

표 2. Stable Diffusion 성능 비교

OpenCL 로 구현한 Stable Diffusion 의 성능을 비교하기 위해 비교군인 Pytorch 모델을 CPU 상 구동한 경우와 OpenCL 의 Naive 한 구현체와 이를 앞서 언급한 최적화 방식을 통해 최적화 했을 때를 비교하였다.

Pytorch 모델의 경우 CPU 에서 구동하는 상황에서도 우수한 성능을 보여주었다. 순수한 추론 시간과 모델을 불러오는 과정을 포함한 추론 시간을 측정하였다. Text Encoder 의 경우 모델을 불러오는 시간을 포함한 추론시간은 8959 milli seconds(ms. 약 8.9 s)가 소요되었고 순수 추론 시간은 2661 ms(약 2.6 s)가 소요되었다. U-Net 의 경우 36237 ms(약 36.2 s) 와 13408 ms(약 13.4 s)가 소요되었다. Decoder 의 경우 14647 ms(약 14.6 s)와 13422 ms(약 13.4 s)가 소요되었다. Stable Diffusion 모델을 로드하지 않은 상태에서 각 모델으로 추론했을 때 59010 ms(약 59 s)가 소요되는 것으로 측정했다.

OpenCL 의 Naive 한 구현의 경우는 Pytorch 와 비교할 수 없을 정도로 느린 결과를 확인할 수 있었다. OpenCL 의 경우 U-Net 모델을 제외한 Text Encoder 와 Decoder 모델은 미리 weight 와 bias 를 메모리에 올린 상태에서 추론 시간을 측정하여 OpenCL 의 연산 시간만을

집중하여 최적화 결과를 비교하였다. Naive 한 Text Encoder 는 순수 추론 시간만 17448 ms(약 17.4 s)가 소요되었다. U-Net 모델의 경우 1055231 ms (약 17 m 35s)가 소요되었고 Decoder 모델의 경우 1744354 ms(약 29m 4s)가 소요되었다.

OpenCL 의 최적화 모델은 2 가지 최적화 방식을 적용한 뒤 성능을 각 최적화 이후의 성능을 측정하였다. Decoder 모델의 경우 im2win 방식의 적용으로 메모리 사용량이 늘어남에 따라 일부분의 weight 와 bias 만을 미리 로드하여 시간 소요를 측정하였다. Text Encoder 모델의 경우 2D Convolution 을 사용하지 않고 Linear 과 행렬곱이 연산을 구성하여 행렬곱 최적화에 대한 시간 소요만을 측정하였다. local memory 및 register 을 통한 행렬곱 최적화를 적용한 후 추론에 8405 ms(약 8.4 s)가 소요되었다. U-Net 모델은 im2col 방식을 적용한 경우 466358 ms(약 7m 46s)가 소요되었고 im2win 방식의 경우 455625 ms(약 7m 35.6s)가 소요되었다. im2win 의 최적화에 행렬곱 최적화를 적용한 경우에는 426683 ms(약 7m 6.6s)가 소요되었다. Decoder 모델의 경우 Text Encoder 와 반대로 2D Convolution 이 연산의 대부분을 차지하여 im2win 최적화에 대한 소요 시간을 측정한 결과 186762 ms(3m 6.7s)가 소요됨을 알 수 있었다.

결과적으로 Pytorch 모델의 추론 시간이 모든 모델에 대하여 OpenCL 구현체보다 성능이 좋았다. 각 모델의 경우 자세히 살펴보면 Text Encoder 의 경우 Pytorch 의 2.6s 보단 느리지만 8.9s 수준으로 성능을 최적화 할 수 있었다. U-Net 의 경우 2D Convolution 의 최적화만으로도 17 분 35 초에서 7 분 35 초로 약 10 분 정도를 최적화 할 수 있었고 여기에 행렬곱 최적화를 통해 29 초를 더 줄일 수 있었다. Decoder 모델은 naive 한 구현체가 29 분 4 초가 소요되었지만 최적화 이후에 3 분 6 초대로 줄어들었다. 상대적으로 차원의 크기가 작은 값을 다루는 Text Encoder 의 경우 Pytorch 모델과 차이가 4 배로 다른 모델에 비해 상대적으로 작았지만 차원의 크기가 큰 값을 연산하는 U-Net 의 경우 약 11.7 배 더 많은 시간이 소요되었고 Decoder 의 경우 12.7 배 더 많은 시간이 소요되었다. 차원의 크기가 크면 그만큼 메모리의 효율성이 중요해져 이에 대한 비용으로 더 큰 시간이 소요되는 것으로 분석할 수 있었다.

제 3 절 정리

본 연구에서 Stable Diffusion 모델을 모바일 환경에 옮겨 최적화하는 과정을 통해 on-device AI 에 대한 실용성을 알 수 있었다. Pytorch 모델의 경우 CPU 에서 좋은 성능을 냈다. 본 연구에서 구현한 OpenCL 의 경우 연산의 최적화 방식만으로 성능을 끌어올리면서 가능성을 엿볼 수 있었다. 이를 통해 추후 Global memory 에 대한 접근을 최소한으로 줄이고 GPU 의 Parallelism 을 최대한으로 활용하면 본 연구보다 나은 성능을 얻을 수 있을 것이라 기대한다. FP32 모델에서 FP16 모델로 양자화하는 방식을 통해 메모리 부족에 대한 해결책을 얻으면서 더욱 빠른 추론을 할 수 있을 것이다. 이 연구가 실질적인 환경에 대한 on-device AI 발전에 기여를 할 수 있기를 바라며 논문을 마친다.

참 고 문 헌

[1] J. A. Baktash and M. Dawodi, Gpt-4: A Review on Advancements and Opportunities in Natural Language Processing. 2023.

[2] “ChatGPT,” 2023. [Online]. Available: https://chat.openai.com/

[3] “Midjourney,” 2023. [Online]. Available: https://www.midjourney.com/

[4] “5 benefits of on-device generative AI,” 2023. [Online]. Available: https://www.qualcomm.com/news/onq/2023/08/5-benefits-of-on-device-generative-ai

[5] “OpenCL,” 2023. [Online]. Available: https://www.khronos.org/opencl/

[6] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, High-Resolution Image Synthesis with Latent Diffusion Models. 2022.

[7] Y.-H. Chen et al., Speed Is All You Need: On-Device Acceleration of Large Diffusion Models via GPU-Aware Optimizations. 2023.

[8] J. Choi et al., Squeezing Large-Scale Diffusion Models for Mobile. 2023.

[9] "World's first on-device demonstration of Stable Diffusion on an Android phone," 2023. [Online]. Available: https://www.qualcomm.com/news/onq/2023/02/worlds-first-on-device-demonstration-of-stable-diffusion-on-android

[10] "Mali-G78," 2023. [Online]. Available: https://developer.arm.com/Processors/Mali-G78

[11] "Exynos 2100," 2023. [Online]. Available: https://semiconductor.samsung.com/processor/mobile-processor/exynos-2100/

[12] R. Liang et al., "Romou: Rapidly Generate High-Performance Tensor Kernels for Mobile GPUs," Feb. 2022. [Online]. Available: https://www.microsoft.com/en-us/research/publication/romou-rapidly-generate-high-performance-tensor-kernels-for-mobile-gpus/

[13] "Performance Ranking," 2023. [Online]. Available: https://ai-benchmark.com/ranking.html

[14] "Detailed Result," 2023. [Online]. Available: https://ai-benchmark.com/ranking_detailed.html

[15] "Stable Diffusion Version 2," 2023. [Online]. Available: https://github.com/Stability-AI/stablediffusion

[16] "OpenCLIP," 2023. [Online]. Available: https://github.com/mlfoundations/open_clip

[17] "Stable Diffusion v2-1-base," 2023. [Online]. Available: https://huggingface.co/stabilityai/stable-diffusion-2-1-base

[18] S. Lu, J. Chu, and X. T. Liu, "Im2win: Memory Efficient Convolution On SIMD Architectures," Sep. 2022. doi: 10.1109/hpec55821.2022.9926408.

[19] A. Vasudevan, A. Anderson, and D. Gregg, Parallel Multi Channel Convolution using General Matrix Multiplication. 2017.

[20] “Low Memory Killer Daemon,” 2023. [Online]. Available:
https://source.android.com/docs/core/perf/lmkd
[21] O. Ronneberger, P. Fischer, and T. Brox, U-Net: Convolutional Networks for Biomedical Image Segmentation. 2015.

Abstract

The Optimization of Stable Diffusion based on the Mobile

Hyeonwoo Koo

Computer Science and Engineering

The Graduate School

Seoul National University

The performance of System on Chip (SoC) for mobile devices is advancing as time progresses. On-Device AI technology has limitations in restricted environments, but it can provide good solutions for security and cost issues. In this study, the Stable Diffusion model was optimized on Samsung's Galaxy S21 Exynos 2100 device. Utilizing the open-source Stable Diffusion v2.1 PyTorch implementation, we developed an Android app using Android NDK, C/C++, and OpenCL. Performance improvements through optimization techniques were compared with the baseline PyTorch model. The study confirmed the practical application of On-Device AI and provided suggestions for additional optimization approaches.

keywords : On-Device AI, OpenCL, Stable Diffusion, Optimization