Parallel Programmeren: Classificatie met Neurale Netwerken

Pieter Dilien, Waldo Wautelet

May 20, 2022

1 Introductie

In dit verslag zal het paralleliseren van een programma dat een afbeelding kan classificeren besproken worden. Eerst zal het originele niet-geparalleliseerde programma bekeken worden. Vervolgens zal er ingegaan worden op de verbeteringen die er aan het originele programma aangebracht zijn om het parallel te laten functioneren. Ten slotte zullen de resultaten van de beide programma's vergeleken worden.

2 Origineel

Eerst en vooral werd bestudeerd hoe het programma juist werkte. Vervolgens bekeken we welke layers het meeste computing time innamen. Dit om te bepalen welke layers nog het meest ruimte voor verbetering hadden. De layertijden van het originele programma zijn te zien op figuur 2.

Het is duidelijk dat layers 1, 3, 6, 10, 14, 18, 19, 21 en 22 reeds erg snel zijn. Deze hebben dus minder marge voor verbetering. Hierdoor keken we naar de overgebleven layers. Het werd snel duidelijk dat deze allemaal gebruikmaken van de "convolution_layer" functie. Het is dan ook deze functie dat gekozen werd om te paralleliseren.

3 Verbeterd

We hebben de functie "convolution_layer" geparalleliseerd door deze over te plaatsen op de GPU. Hier waren enkele moeilijke omzettingen nodig waarover goed moest worden nagedacht. Zo was het dat we een rand met 0 rond onze input_data moesten krijgen en dit origineel ook in dezelfde kernel uitvoerde. Dit was achteraf echter geen goed idee dus hier hebben we een aparte kernel voor aangemaakt.

Nu werken we dus met 2 aparte kernels, de eerste kernel is om onze input_data in de zeropad buffer te steken. Dit is een zeer korte en eenvoudige kernel die enkel moet lezen van input_data en schrijven naar zeropad.

Voordat we de kernel kunnen gebruiken moeten we eerst alle gebruikte (gedeelde) buffers vullen. De zeropad buffer word zeer simpel gevuld met allemaal 0, de andere buffers worden gevuld met meegegeven data omdat de GPU hier data van moet lezen.

De tweede kernel is de kernel die effectief de convolutielaag gaat uitrekenen. We hebben ervoor gekozen om zoveel mogelijk te paralleliseren en dus hebben we de gpu opgedeeld in feature_size * feature_size * input_depth work-items. Door dit te doen moeten we slechts 1 for-loop uitvoeren in onze tweede kernel. We hebben ook gebruik gemaakt van een atomische som, dit was nodig omdat het kan zijn dat 2 work-items tegelijk een '+='-operatie willen uitvoeren en dit noemt men een race-conditie. Hier kan de uitkomst dus niet correct zijn. Door deze atomische add word dit vermeden en is de uitkomst dus altijd correct.

Ten slotte lezen we alle data uit deze gedeelde buffer en steken we deze in het geheugen van de CPU.

4 Resultaten

Het verbeterde programma is duidelijk sneller dan het originele, dit is zichtbaar in figuur 1. De totale computing time werd verlaagd van gemiddeld 17 seconden naar gemiddeld 2.5 seconden.

Daarnaast is duidelijk te zien in figuur 2 dat het opzet om layers 2, 4, 5, 7, 8, 9, 11, 12, 13, 15, 16 en 17 - die allemaal gebruikmaken van de "convolution_layer" functie - te versnellen geslaagd is.

Het is ook goed te zien dat al de andere layers die nog steeds volledig op de CPU uitgevoerd worden en dus niet geparalleliseerd zijn nog steeds dezelfde tijd hebben. Specifiek voor de laatste 3 layers is er nog ruimte voor versnelling. Dit zou eventueel mogelijk geweest zijn door de "dense" functie te paralleliseren.

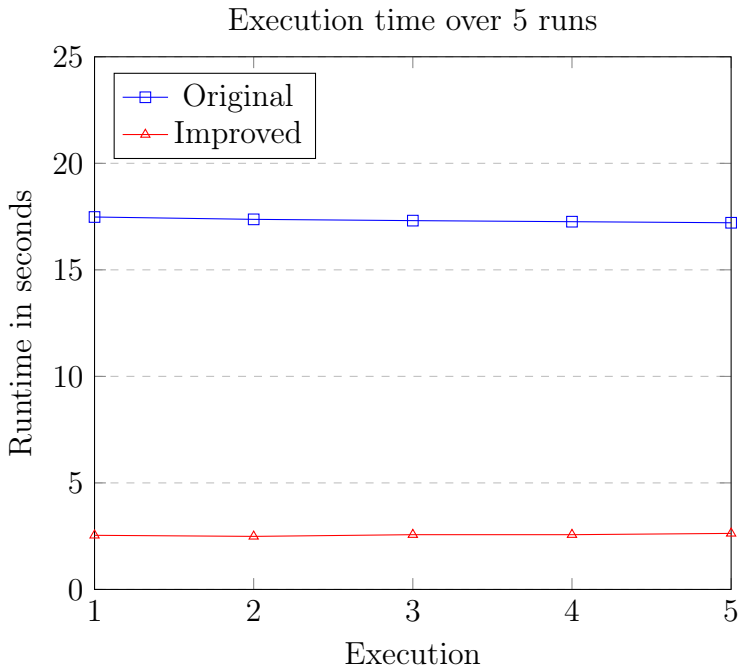

Figure 1: Execution time of original and improved program

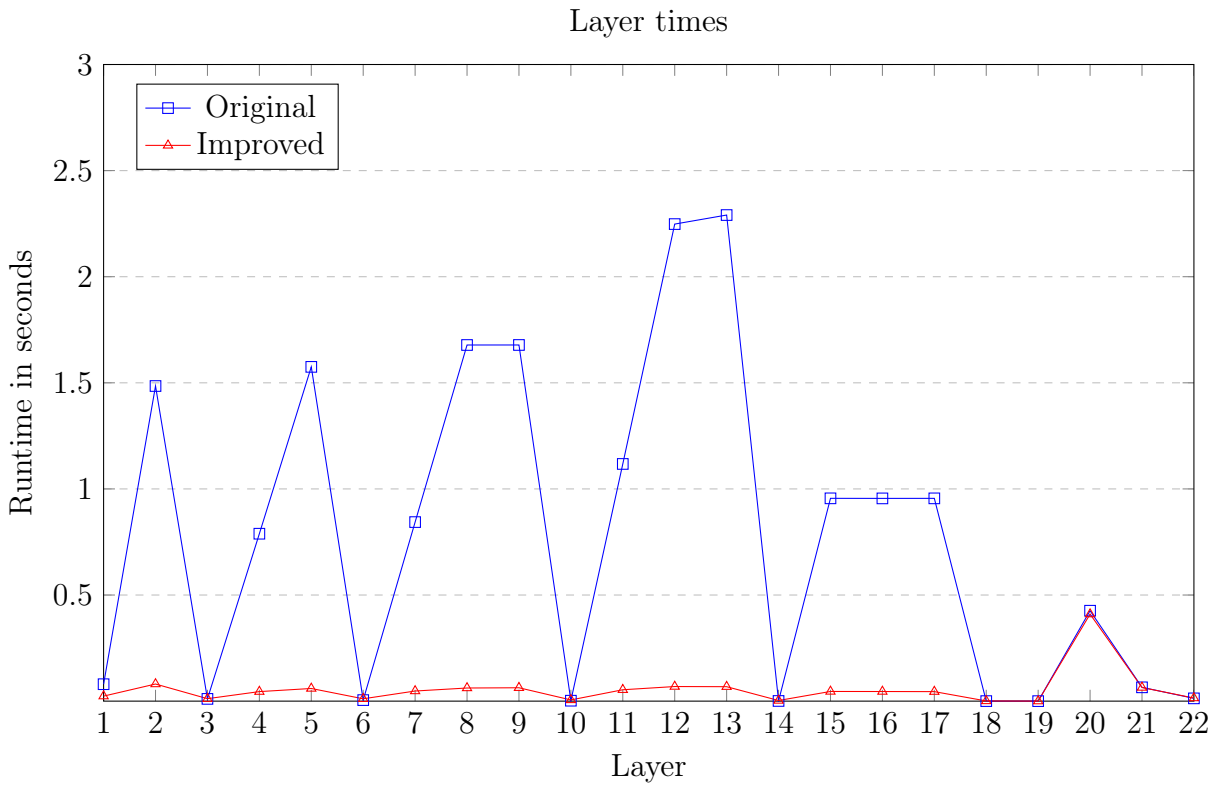

Figure 2: Layer times of original and improved program

5 Conclusie

Ondanks het succesvol versnellen van het programma, kregen we wel nog een fout resultaat. We zouden "tabby_cat" moeten krijgen, maar krijgen in plaats daarvan steeds het resultaat "buckle". We hebben het grootste deel van het labo gezocht naar de fout(en) waardoor we steeds een incorrect resultaat kregen, maar vonden deze tevergeefs niet.