UNIVERSITAS INDONESIA

DEEP LEARNING INFERENCE PADA ANDROID GPU MENGGUNAKAN OPENCL DAN VULKAN

TUGAS AKHIR

TSESAR RIZQI PRADANA
1406543725

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI ILMU KOMPUTER
DEPOK
JANUARI 2018

UNIVERSITAS INDONESIA

DEEP LEARNING INFERENCE PADA ANDROID GPU MENGGUNAKAN OPENCL DAN VULKAN

TUGAS AKHIR
Diajukan sebagai salah satu syarat untuk memperoleh gelar
Sarjana Komputer

TSESAR RIZQI PRADANA
1406543725

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI ILMU KOMPUTER
DEPOK
JANUARI 2018

HALAMAN PERSETUJUAN

Judul	:	Deep Learning Inference pada Android GPU Menggunakan OpenCL dan Vulkan
Nama	:	Tsesar Rizqi Pradana
NPM	:	1406543725

Laporan Tugas Akhir ini telah diperiksa dan disetujui.

20 Januari 2018

Prof. T. Basaruddin

Pembimbing Tugas Akhir

HALAMAN PERNYATAAN ORISINALITAS

Tugas Akhir ini adalah hasil karya saya sendiri,
dan semua sumber baik yang dikutip maupun dirujuk
telah saya nyatakan dengan benar.

Nama	:	**Tsesar Rizqi Pradana**
NPM	:	**1406543725**
Tanda Tangan	:	
Tanggal	:	**20 Januari 2018**

HALAMAN PENGESAHAN

Tugas Akhir ini diajukan oleh	:	
Nama	:	Tsesar Rizqi Pradana
NPM	:	1406543725
Program Studi	:	Ilmu Komputer
Judul Tugas Akhir	:	Optimisasi Inference pada Convolutional Neural Network melalui Operation Graph

Telah berhasil dipertahankan di hadapan Dewan Penguji dan diterima sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Sarjana Komputer pada Program Studi Ilmu Komputer, Fakultas Ilmu Komputer, Universitas Indonesia.

DEWAN PENGUJI

Pembimbing	:	Prof. T. Basaruddin	( )
Penguji	:		( )
Penguji	:		( )
Penguji	:		( )

Ditetapkan di	:	Depok
Tanggal	:	20 Januari 2018

KATA PENGANTAR

Puji dan syukur Penulis panjatkan kepada Tuhan Yang Maha Esa. Berkat Rahmat-Nya Tugas Akhir yang berjudul ”Deep Learning Inference pada Android GPU Menggunakan OpenCL dan Vulkan” ini dapat diselesaikan.

Banyak kendala yang dialami Penulis dalam menyelesaikan Tugas Akhir ini. Namun, Penulis dapat mengatasinya berkat bantuan dari dosen pembimbing, orang tua, teman-teman, dan pihak-pihak lainnya.

Penelitian ini disusun dalam waktu yang cukup singkat sehingga masih terdapat banyak kekurangan baik itu pada konten penelitian maupun pada penulisan. Penulis mengharapkan kritik dan saran dari pembaca sebagai bahan pembelajaran bagi Penulis untuk menyusun karya-karya di kemudian hari.

Melalui kata pengantar ini Penulis juga ingin mengucapkan banyak terimakasih kepada pihak-pihak yang telah membantu penyelesaian Tugas Akhir ini, antara lain:

1. Orang tua yang telah memberikan dukungan moral dan materiil,
2. Prof. T. Basaruddin sebagai Pembimbing I,
3. Bapak Risman Adnan sebagai Pembimbing II,
4. Ryorda Triaptahadi sebagai rekan penelitian, dan
5. Teman-teman semua yang telah memberikan dukungan moral.

Semoga pihak-pihak yang telah disebutkan mendapatkan balasan dari Tuhan Yang Maha Esa atas bantuan mereka dalam penyusunan Tugas Akhir ini. Penulis berharap Tugas Akhir ini dapat memberikan manfaat yang positif kepada berbagai pihak.

Depok, 20 Desember 2017

Tsesar Rizqi Pradana

HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI TUGAS AKHIR UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Universitas Indonesia, saya yang bertanda tangan di bawah ini:

Nama	:	Tsesar Rizqi Pradana
NPM	:	1406543725
Program Studi	:	Ilmu Komputer
Fakultas	:	Ilmu Komputer
Jenis Karya	:	Tugas Akhir

demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Universitas Indonesia **Hak Bebas Royalti Noneksklusif (Non-exclusive Royalty Free Right)** atas karya ilmiah saya yang berjudul:

Deep Learning Inference pada Android GPU Menggunakan OpenCL dan Vulkan

beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Universitas Indonesia berhak menyimpan, mengalihmedia/formatkan, mengelola dalam bentuk pangkalan data (database), merawat, dan memublikasikan tugas akhir saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta.

Demikian pernyatan ini saya buat dengan sebenarnya.

Dibuat di : Depok
Pada tanggal : 20 Januari 2018

Yang menyatakan

(Tsesar Rizqi Pradana)

ABSTRAK

Nama	:	Tsesar Rizqi Pradana
Program Studi	:	Ilmu Komputer
Judul	:	Deep Learning Inference pada Android GPU Menggunakan OpenCL dan Vulkan

Deep Learning has been widely used in many Artificial Intelligence (AI) application because of its high accuracy. With its impressive advances in AI fields, many developers have started to create innovations on Deep Learning such as creating the possibility to to run Deep Learning on mobile device. Because of its high computational cost and the ability of today's mobile device, it is only possible to do the inference step of Deep Learning. Even for inference, current mobile Deep Learning libraries are not optimal yet because of the lack of support from developers. Popular libraries like Tensorflow Mobile or Tensorflow Lite only supports Deep Learning inference on mobile CPU. In this paper we try to create an accelerator for mobile Deep Learning inference by implementing OpenCL code to support Deep Learning inference on mobile GPU. We want to evaluate the effects of applying the accelerator on mobile Deep Learning inference performance with expectation that it will produce a good performance. We also want to compare the accelerators performance to the original CPU implementations performance.

We have implemented OpenCL code on some Deep Learning inference operations including matrix multiplication, matrix convolution, matrix transpose, and vector addition. We test our implementation using Tensorflow Lite demo application on Android devices with some different Convolutional Neural Network models including Inception, LeNet, and MobileNet. The result shows that Deep Learning Inference operations on OpenCL can achieve much better running time for large enough matrix. However, our accelerator could perform much slower than the original Tensorflow Lite functions for small matrix. This is caused by the time taken for OpenCL setup operations like memory allocation and load variable values to GPU memory are much longer than the actual inference operations themself. This can be seen from the similarity of running time on each operations for small matrix, which indicate the bottleneck on OpenCL setup operations. Further optimization on the OpenCL kernels has been applied and it still does not omit the bottleneck.

Kata Kunci:

@todo
Tuliskan kata kunci yang berhubungan dengan laporan disini

ABSTRACT

Name	:	Tsesar Rizqi Pradana
Program	:	Computer Science
Title	:	Deep Learning Inference on Android GPU with OpenCL and Vulkan

Deep Learning has been widely used in many Artificial Intelligence (AI) application because of its high accuracy. With its impressive advances in AI fields, many developers have started to create innovations on Deep Learning such as creating the possibility to to run Deep Learning on mobile device. Because of its high computational cost and the ability of today's mobile device, it is only possible to do the inference step of Deep Learning. Even for inference, current mobile Deep Learning libraries are not optimal yet because of the lack of support from developers. Popular libraries like Tensorflow Mobile or Tensorflow Lite only supports Deep Learning inference on mobile CPU. In this paper we try to create an accelerator for mobile Deep Learning inference by implementing OpenCL code to support Deep Learning inference on mobile GPU. We want to evaluate the effects of applying the accelerator on mobile Deep Learning inference performance with expectation that it will produce a good performance. We also want to compare the accelerators performance to the original CPU implementations performance.

We have implemented OpenCL code on some Deep Learning inference operations including matrix multiplication, matrix convolution, matrix transpose, and vector addition. We test our implementation using Tensorflow Lite demo application on Android devices with some different Convolutional Neural Network models including Inception, LeNet, and MobileNet. The result shows that Deep Learning Inference operations on OpenCL can achieve much better running time for large enough matrix. However, our accelerator could perform much slower than the original Tensorflow Lite functions for small matrix. This is caused by the time taken for OpenCL setup operations like memory allocation and load variable values to GPU memory are much longer than the actual inference operations themself. This can be seen from the similarity of running time on each operations for small matrix, which indicate the bottleneck on OpenCL setup operations. Further optimization on the OpenCL kernels has been applied and it still does not omit the bottleneck.

Keywords:

@todo
Write up keywords about your report here.

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB 1
PENDAHULUAN

Karya tulis yang berjudul ”Deep Learning Inference pada Android GPU Menggunakan OpenCL dan Vulkan” ini didahului dengan pembahasan mengenai latarbelakang penelitian, permasalahan yang ingin diselesaikan, tujuan penelitian, serta metodologi dari penelitian.

1.1 Latar Belakang

Deep Learning merupakan salah satu cabang dari Machine Learning yang dianggap memiliki performa lebih baik daripada algoritma Machine Learning konvensional. Deep Learning memanfaatkan *Neural Network* untuk melakukan prediksi label atau skor dari data [1]. Arsitektur *Neural Network* pada Deep Learning memiliki struktur yang lebih kompleks dengan lebih dari satu hidden layer. Deep Learning bekerja dengan cara melakukan generalisasi dari sekumpulan pengalaman yang telah dipelajari. Proses mempelajari pengalaman dilakukan pada tahap training (latihan) menggunakan sekumpulan data hasil observasi. Neural Netowrk yang telah dilatih dapat digunakan untuk melakukan prediksi label atau skor dari data baru.

Popularitas Deep Learning dalam aplikasi Artificial Intelligence saat ini mulai meningkat pesat. Banyak inovasi di bidang Deep Learning yang mulai bermunculan. Salah satu inovasi yang menarik adalah penerapan Deep Learning pada Android sehingga memungkinkan aplikasi Artificial Intelligence dengan akurasi tinggi dapat diakses melalui perangkat kecil yang mudah dibawa kemana saja. Tesorflow [7] merupakan contoh *Deep Learning framework* yang menyediakan dukungan untuk Android melalui *library* Tensorflow Mobile dan Tensorflow Lite. Penerapan Deep Learning pada perangkat *mobile* memiliki tantangan tersendiri. Deep Learning dikenal memiliki beban komputasi yang sangat besar karena mengandung sangat banyak operasi-operasi matriks. Dengan kemampuan perangkat *mobile* yang ada saat ini, Deep Learning hanya mungkin digunakan untuk melakukan *inference* saja.

Meskipun Deep Learning sudah dapat diterapkan pada perangkat mobile, saat ini dukungan yang diberikan masih sedikit. Mayoritas pengembang masih berfokus pada perangkat desktop yang memang memiliki potensi lebih tinggi untuk *Deep Learning* . Sebagai contoh, pada perangkat PC Deep Learning inference maupun

training dapat dijalankan dengan cepat menggunakan GPU. Hampir semua framework Deep Learning memiliki dukungan untuk penggunaan GPU. Pada perangkat mobile, dukungan untuk penggunaan GPU sangat sulit ditemui. Framework populer seperti Tensorflow pun hanya menyediakan dukungan untuk inference menggunakan CPU. Padahal Mobile GPU berpotensi dapat meningkatkan performa inference seperti halnya GPU pada PC.

Hal ini menarik minat penulis untuk melaksanakan penelitian ini. Penulis mencoba menerapkan pemrograman GPU pada operasi-operasi *Deep Learning inference* dengan hipotesis bahwa GPU dapat membantu meningkatkan performa Deep Learning inference pada perangkat mobile. Alasan pertama, GPU memiliki sangat banyak unit pemrosesan yang dapat mengeksekusi suatu pekerjaan secara paralel [18]. Meskipun secara individu unit pemrosesan tersebut lebih lemah dari unit pemrosesan pada CPU, dengan teknik paralelisasi suatu pekerjaan besar dapat dibagi ke banyak unit pemrosesan sehingga secara keseluruhan komputasi dapat berjalan lebih cepat. Kedua, operasi-operasi *Deep Learning inference* merupakan operasi-operasi matriks [17], sehingga berpotensi besar untuk diakselerasi dengan pemrograman paralel. Sebagai contoh, pada operasi penjumlahan matriks, masing-masing penjumlahan antara dua elemen dari dua matriks pada baris dan kolom yang sama dapat diproses oleh suatu unit pemrosesan tersendiri secara independen, sehingga penjumlahan semua elemen dilakukan secara paralel.

Untuk menerapkan penggunaan GPU pada operasi-operasi *Deep Learning inference* penulis menggunakan API OpenCL dan Vulkan. Keduanya merupakan API dalam bahasa C/C++, mendukung pemrograman paralel, dan dapat digunakan untuk *multi-device*. Selain membandingkan inference antara CPU dan GPU, peulis juga tertarik membandingkan kedua API ini untuk melihat apakah terdapat pengaruh dari driver masing-masing API terhadap performa inference. Penulis menggunakan Tensorflow beserta Tensorflow Lite sebagai framework Deep Learning. Tensorflow dan Tensorflow Lite merupakan *framework Deep Learning* paling populer yang didukung oleh Google dan memiliki fitur untuk penggunaan DNN, CNN, dan RNN. Tensorflow sendiri diimplementasikan dengan bahasa C/C++, sehingga kompatibel dengan OpenCL dan Vulkan.

1.2 Permasalahan

Pada bagian ini akan dijelaskan mengenai definisi permasalahan yang Penulis hadapi dan ingin diselesaikan serta asumsi dan batasan yang digunakan dalam menyelesaikannya.

1.2.1 Definisi Permasalahan

Berikut adalah permasalahan-permasalahan yang mendorong dilaksanakannya penelitian ini.

1. Apakah OpenCL dan Vulkan dapat diterapkan pada Tensorflow Lite untuk Deep Learning inference menggunakan *mobile* GPU?
2. Bagaimana pengaruh penerapan pemrograman GPU untuk *inference* pada model-model *Deep Learning* pada perangkat *mobile* ?
3. Bagaimana perbandingan kecepatan, penggunaan memori, dan penggunaan baterai pada *Deep Learning inference* menggunakan CPU dan GPU?
4. Operasi-operasi *Deep Learning inference* apa saja yang lebih baik dijalankan pada *mobile* GPU?
5. Apakah terdapat perbedaan performa dari OpenCL dan Vulkan dalam melakukan komputasi pada GPU?

1.2.2 Batasan Permasalahan

Batasan-batasan penelitian pada Tugas Akhir ini antara lain:

1. Implementasi OpenCL dan Vulkan hanya berlaku untuk perangkat *Android* saja.
2. OpenCL dan Vulkan hanya diterapkan pada beberapa operasi *Deep Learning inference* pada kode sumber Tensorflow Lite, antara lain perkalian matriks dan konvolusi matriks.
3. Pengujian hanya dilakukan terhadap beberapa arsitektur *Convolutional Neural Network* , antara lain Inception, LeNet, dan MobileNet, yang telah mencakup operasi konvolusi dan perkalian matriks.
4. Penulis mengasumsikan bahwa perangkat yang digunakan hanya menggunakan sumber daya multi-core CPU dan GPU beserta memorinya, terlepas dari dorongan performa yang berasal dari perangkat tambahan.

1.3 Tujuan

Tujuan dari penelitian ini adalah sebagai berikut.

1. Mengimplementasikan OpenCL dan Vulkan code untuk penggunaan mobile GPU pada operasi-operasi Deep Learning inference.
2. Mengetahui pengaruh penggunaan mobile GPU melalui OpenCL dan Vulkan pada proses inference pada model-model Deep Learning.
3. Membandingkan performa, penggunaan memori, dan penggunaan baterai pada proses inference pada CPU dan GPU.
4. Mengetahui operasi-operasi Deep Learning inference pada yang lebih baik bila dijalankan pada GPU melalui OpenCL dan Vulkan.
5. Mengetahui perbedaan pengaruh masing-masing API (OpenCL dan Vulkan) terhadap performa inference.

1.4 Metodologi Penelitian

Metodologi penelitian pada Tugas Akhir ini adalah sebagai berikut.

1. Menentukan Tensorflow Lite dan OpenCL sebagai *framework* dan API yang digunakan untuk melakukan penelitian.
2. Mempelajari kode sumber Tensorflow Lite.
3. Mempelajari OpenCL beserta teknik pemrograman GPU.
4. Mengimplementasikan OpenCL code untuk beberapa operasi *Deep Learning inference* .
5. Melakukan eksperimen menggunakan hasil implementasi.
6. Mengoptimalkan implementasi sebelumnya.
7. Melakukan eksperimen menggunakan hasil implementasi yang sudah dioptimalkan.
8. Melaporkan hasil eksperimen sebagai karya tulis.

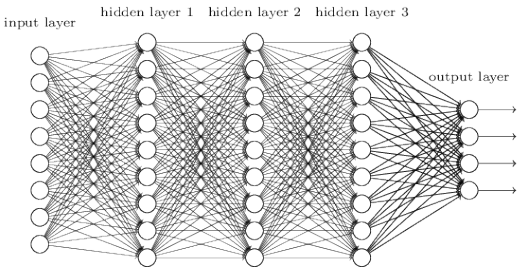
chapterLANDASAN TEORI Bagian ini menjelaskan teori-teori yang digunakan dalam penelitian. Teori yang dimaksud adalah pengetahuan yang terkait dengan pelaksanaan penelitian.

1.5 Deep Learning

Deep Learning merupakan algoritma *Machine Learning* yang memanfaatkan arsitektur *Neural Network* . Cara kerja arsitektur ini terinspirasi oleh struktur otak manusia yang tersusun dari neuron-neuron. Berbeda dengan Artificial Neural Network biasa, Neural Network pada Deep Learning memiliki struktur yang lebih kompleks, terdiri dari lebih dari satu hidden layer. Ide utama dari Deep Learning adalah bagaimana komputer dapat belajar dari pengalaman dengan cara melatih *Neural Network* menggunakan data-data yang berjumlah besar. *Deep Learning* merupakan salah satu contoh dari *Supervised Learning* yang berarti *Neural Network* dilatih menggunakan data yang telah diketahui labelnya. *Neural Network* yang telah dilatih dapat digunakan untuk menentukan label atau skor dari data-data baru [19].

Deep Learning dikenal memiliki akurasi yang lebih tinggi dibandingkan algoritma Machine Learning konvensional karena kompeksitas Neural network yang digunakan. Akurasi dari prediksi/inference bergantung pada jenis arsitektur *Neural Network* , kualitas dan kuantitas data yang digunakan untuk melatih, serta jenis permasalahan yang diselesaikan. Pada Deep Learning terdapat berbagai jenis arsitektur. Tiga arsitektur yang populer adalah Deep Neural Network, Convolutional Neural Network, dan Recurrent Neural Network. Deep Neural Network. Masing-masing arsitektur memiliki kelebihan dan kekurangan masing-masing. Untuk setiap permasalahan perlu dipilih arsitektur yang tepat agar mendapatkan akurasi maksimal.

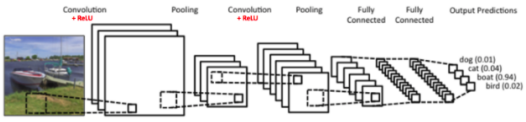
Deep Neural Network (DNN) merupakan jenis arsitektur Deep Learning yang paling sederhana. DNN merupakan *Neural Network* yang terdiri dari satu input layer, satu atau lebih hidden layer, dan satu output layer. Layer-layer pada arsitektur ini merupakan fully connected layer. Setiap layer tersusun dari node-node yang masing-masing terhubung dengan semua node dari layer-layer tetangganya. Nilai dari suatu node pada layer ke-l merupakan hasil penerapan suatu activation function terhadap perkalian dari node-node pada layer ke-(l-1) dengan weight parameter tertentu [17]. Gambar 1.1 menunjukkan contoh arsitektur DNN.



Gambar 1.1: Contoh Arsitektur Deep Neural Network.

Karena arsitekturnya yang sederhana, DNN pada banyak kasus memiliki performa yang tidak sebaik arsitektur lain seperti CNN atau RNN. DNN lebih tepat digunakan untuk melakukan tugas-tugas prediksi sederhana. Fitur-fitur dari data harus sudah ditentukan terlebih dahulu apabila ingin melakukan training dan inference pada DNN. Beban komputasi DNN lebih ringan dari arsitektur lain seperti CNN.

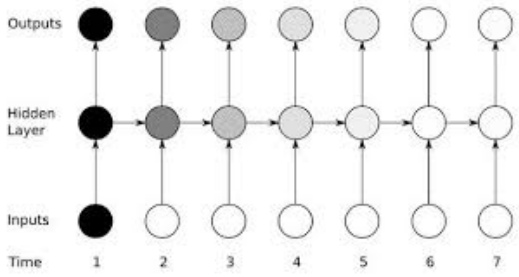
Convolutional Neural Network (CNN) merupakan arsitektur *Deep Learning* yang sering digunakan dalam bidang pengolahan citra. CNN pada umumnya terdiri dari beberapa convolution layer, pooling layer, dan fully-connected layer. Convolution layer dan Pooling layer merupakan jenis layer pada CNN yang tidak terdapat pada arsitektur-arsitektur lain. Convolution layer biasanya berfungsi untuk mengekstrak fitur-fitur dari data yang akan diprediksi secara otomatis. Pada DNN fitur-fitur dari data harus ditentukan sendiri secara manual. Pada convolution layer terjadi operasi konvolusi terhadap input menggunakan suatu kernel tertentu. Pooling layer berfungsi untuk mereduksi ukuran data dengan cara melakukan sampling. Data yang direduksi biasanya adalah keluaran dari convolution layer [17]. Fully-connected layer biasanya diletakkan pada bagian akhir arsitektur CNN. layer ini berfungsi untuk melakukan prediksi, sama seperti DNN, dengan menggunakan fitur-fitur hasil ekstraksi convolution layer. Gambar 1.2 merupakan contoh arsitektur CNN.



Gambar 1.2: Contoh Arsitektur *Convolutional Neural Network* .

Recurrent Neural Network (RNN) merupakan arsitektur Deep Learning yang dapat digunakan untuk jenis data yang bersifat sekuensial. Arsitektur ini terdiri dari input layer, hidden layer, dan output layer yang mengandung loop. Hasil prediksi dari output layer pada suatu waktu t digunakan untuk mengubah parameter

dari layer-layer sebelumnya untuk melakukan prediksi pada waktu *t+1*. Jadi dapat dikatakan bahwa output dari suatu layer pada suatu waktu bergantung pada output dari layer itu sendiri pada waktu-waktu sebelumnya [19]. Gambar 1.3 merupakan contoh arsitektur RNN.



Gambar 1.3: Contoh Arsitektur Recurrent Neural Netowork.

RNN sering digunakan dalam bidang Natural Language Processing (NLP) karena tugas-tugas pada NLP pada umumnya melibatkan pengolahan teks atau suara dengan aturan bahasa, sehingga datanya bersifat sekuensial.

Dengan melihat contoh-contoh arsitektur *Deep Learning* di atas, dapat diketahui bahwa Deep Learning memiliki beban komputasi yang cukup besar, terutama pada CNN, karena melibatkan banyak parameter dan banyak layer. Oleh karena itu, penggunaan CPU saja dirasa kurang cukup dalam menjalankan operasi-operasi pada Deep Learning.

1.6 Deep Learning Training dan Inference

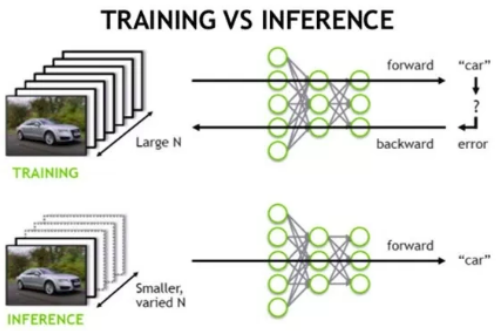
Sama seperti *Neural Network* biasa, arsitektur *Neural Network* dalam Deep Learning memiliki parameter *weight* pada setiap layer yang digunakan untuk menghitung skor dari input pada suatu *layer* dan meneruskannya ke *layer* berikutnya. Deep Learning terdiri dari dua tahap yaitu training dan inference. Pada tahap training, model *Neural Network* dilatih menggunakan sekumpulan data. Dalam proses ini weight parameter terus-menerus memperbarui nilainya hingga pada akhirnya konvergen. Nilai parameter yang konvergen menandakan weight tersebut telah merepresentasikan data-data training [17].

Untuk mengawali proses *training* , *weight* awal diberikan terlebih dahulu kepada model. Biasanya *weight* awal ini merupakan angka-angka acak berdasarkan suatu distribusi tertentu. Selanjutnya, model dengan *weight* awal yang belum tersebut digunakan untuk melakukan prediksi label atau skor dari data-data training dengan cara melakukan *forward pass* di sepanjang model, mulai dari *layer* pertama (input

layer) hingga mencapai *layer* terakhir (*output layer*). Hasil prediksi pada output layer kemudian dibandingkan dengan label aslinya. Nilai *error* dari prediksi dapat dihitung menggunakan fungsi error tertentu, misalnya *Mean Square Error*. Informasi *error* ini kemudian dikirim kembali ke *layer* -*layer* sebelumnya dan akan digunakan oleh *layer* -*layer* terebut untuk memperbarui parameter *weight* . *Weight* dapat diperbarui menggunakan algoritma atau fungsi pembaruan *weight* tertentu. Proses mengembalikan *error* ke belakang ini disebut *back propagation* [20].

Setelah *training* selesai, model *Neural Network* dapat digunakan untuk melakukan proses *inference* . Tahap *inference* merupakan tahap dimana model yang telah dilatih digunakan untuk melakukan prediksi. Proses prediksi ini sama seperti prediksi ketika melakukan *training* , yaitu dengan melakukan *forward-pass* terhadap input data mulai dari *input layer* hingga *output layer* pada model. Letak perbedaan proses *inference* dan *training* adalah pada tahap *back-propagation*. Setelah melakukan prediksi pada proses *inference* , tidak lagi dilakukan *back-propagation* terhadap *error* hasil prediksi seperti pada proses *training* . Tujuan dari *inference* adalah hanyalah mendapatkan hasil prediksi skor atau label yang dikeluarkan oleh model yang telah dilatih [20].

Gambar 1.4 menunjukkan perbedaan proses *training* dan *inference* pada Deep Learning. Terlihat bahwa pada tahap *training* terjadi pemrosesan data pada dua arah (forward-pass dan back-propagation), sedangkan pada *inference* hanya terjadi satu arah (forward-pass saja).



Gambar 1.4: Perbedaan proses *training* dan *inference* pada *Deep Learning* .

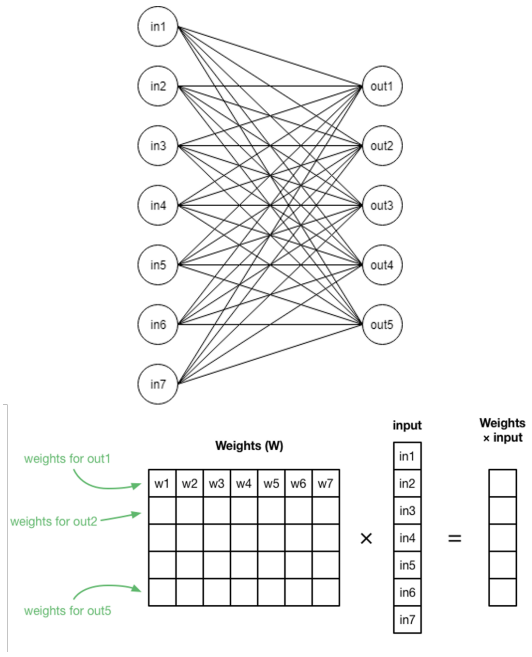
1.7 Operasi-operasi Deep Learning Inference

Tahap inference pada Deep Learning melibatakan banyak operasi matriks. Matriks yang dioperasikan biasanya adalah input data dan weight atau kernel pada suatu layer. Perkalian matriks-vector dan konvolusi merupakan dua operasi pada Deep Learning Inference yang memiliki kompleksitas tinggi. Berikut adalah penjelasan

lebih lanjut mengenai dua operasi tersebut.

1.7.1 Perkalian Matriks-Vector

Operasi perkalian matriks-vector pada Deep Learning inference merupakan salah satu operasi yang memiliki beban komputasi terbesar. Perkalian matriks-vector dapat terjadi pada fully-connected layer. Pada layer ke-*l*, parameter *weight* disimpan dalam bentuk matriks yang berukuran *MxN* dimana *M* adalah banyaknya node pada layer ke-l dan *N* adalah banyaknya node pada layer ke-(l-1) [17]. Baris ke-i pada *weight* matriks dari layer ke-l tersebut merupakan weight dari node ke-i pada layer ke-l. Untuk memperoleh nilai node-node pada layer ke-l, matrix *MxN* tersebut dikalikan dengan vector sepanjang *N* yang berisi nilai-nilai node pada layer ke-(l-1). Hasilnya adalah vector sepanjang *M*. Jika ada, bias ditambahkan kepada vector tersebut. Selanjutnya, activation function diterapkan terhadap setiap elemen vector sehingga menghasilkan vector sepanjang *M* yang merupakan nilai node-node pada layer ke-l. Gambar 1.5 adalah ilustrasi dari operasi perkalian matriks-vector pada fully-connected layer.



Gambar 1.5: Ilustrasi perkalian matriks dengan vector pada fully-connected layer saat mengalikan weight dengan input.

Misalkan matrix weight untuk layer ke-l adalah *Mat* dan vector input dari layer ke-(l-1) adalah *Vec*, kompleksitas dari operasi perkalian matriks pada layer ke-l dapat dihitung sebagai berikut.

$$
\begin{aligned}
dot(Mat[row = 1], Vec[col = 1]) &= O(N) \\
dot(Mat[row = 2], Vec[col = 1]) &= O(N) \\
&\vdots \\
dot(Mat[row = M], Vec[col = 1]) &= O(N) \\
\hline
Totalkompelksitas &= M * O(N) \\
&= O(MN) \qquad (1.1)
\end{aligned}
$$

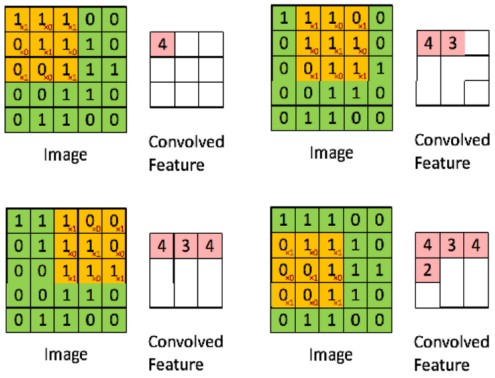
1.7.2 Konvolusi Matriks

Operasi konvolusi terjadi pada convolution layer dari CNN. Konvolusi merupakan penerapan filter/kernel terhadap input dengan mengkonvolusikan kernel tersebut [17]. Proses konvolusi dapat menggunakan satu atau lebih filter. Input, output, dan filter pada convolution layer masing-masing merupakan matriks 3 dimensi yang merepresentasikan lebar, tinggi, dan kedalaman (channel). Ukuran dari input, filter, dan output pada convolution layer memilliki beberapa aturan sebagai berikut.

1. Tinggi dan lebar filter selalu lebih kecil atau sama dengan tinggi dan lebar intput.
2. Kedalaman input selalu sama dengan kedalaman filter.
3. Banyaknya filter sama dengan kedalaman output.
4. Tinggi dan lebar output selalu lebih kecil atau sama dengan tinggi dan lebar intput.

Pada proses konvolusi, posisi filter dimulai dari ujung kiri atas dari input. Filter kemudian bergeser ke kanan dan kebawah dengan stride (jarak geser) yang ditentukan. Pada suatu posisi filter, masing-masing elemen pada filter dikalikan dengan elemen-elemen matriks input yang bersesuaian dengan posisi filter tersebut (seperti dot product pada vector). Hasil dari satu kali operasi tersebut merupakan satu elemen dari matriks output pada posisi yang bersesuaian. Sehingga, setelah proses konvolusi selesai untuk satu filter, akan terbentuk satu lapis output dua dimensi. Jika semua filter sudah diterapkan, maka akan terbentuk output tiga dimensi dengan kedalaman sesuai dengan banyaknya filter. Gambar 1.6 menunjukkan contoh

operasi konvolusi dengan input berukuran 5x5x1, filter berukuran 3x3x1, dan nilai stride 1.



Gambar 1.6: Contoh Operasi Konvolusi. Image merupakan gambar masukan, convolved feature merupakan keluaran dari konvolusi.

Pada operasi konvolusi dengan input berukuran MxNxD, filter berukuran M'xN'xD, banyak filter adalah K, kompleksitas dapat dihitung sebagai berikut. Pada suatu posisi filter, kompleksitas dot product untuk menghasilkan satu elemen output adalah sebagai berikut.

$$Kompleksitasdotproduct = O(M'N'D) \tag{1.2}$$

Kernel bergeser sebanyak O(N) ke kanan dikalikan dengan sebanyak O(M) ke bawah. Diperoleh komleksitas geser sebagai berikut.

$$Kompleksitasgeserkernel = O(MN) \tag{1.3}$$

Menggunakan Persamaan 1.2 dan Persamaan 1.3, diperoleh total kompleksitas sebagai berkut.

$$\begin{aligned} Totalompleksitas &= kompleksitasdotproduct * kompleksitasgeserkernel \\ &= O(MM'NN'DK) \end{aligned} \tag{1.4}$$

Dengan kompleksitas *O(MM'NN'DK)*, konvolusi merupakan operasi paling mahal jika dibandingkan operasi pada layer-layer lain.

1.8 Tensorflow

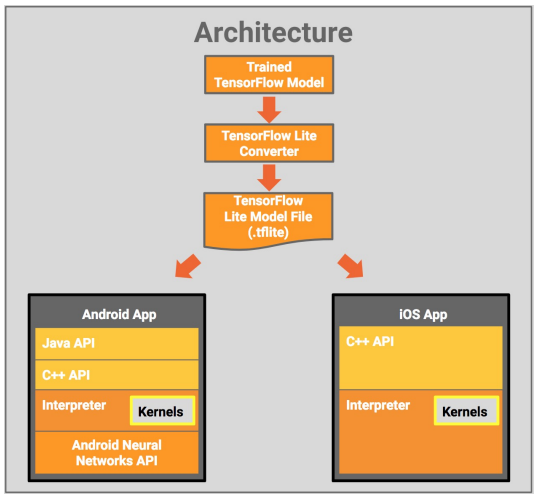
Tensorflow merupakan perangkat lunak *open-source* yang digunakan untuk komputasi numerik menggunakan representasi graf yang membentuk serangkaian op-

erasi [7]. Node pada graf merepresentasikan suatu operasi sedangkan edge mereperesentasikan tensor atau multidimensional array yang merupakan input atau output dari suatu operasi. Tensorflow dikembangkan oleh Google untuk mendukung penelitian pada bidang *Machine Learning* dan Deep Learning. Salah satu fitur menarik dari Tensorflow adalah adanya dukungan untuk Deep Learning inference pada perangkat *mobile* melalui *library* Tensorflow Mobile dan Tensorflow Lite. Tensorflow dan Tensorflow Lite diimplementasikan menggunakan bahasa C/C++ dengan menyediakan API dalam bahasa C++, Python, dan Java.

1.9 Tensorflow Lite

Tensorflow Lite merupakan salah satu library dari Tensorflow yang dapat digunakan untuk *Deep Learning inference* pada perangkat mobile [21]. Selain Tensorflow Lite, terdapat library sejenis yaitu Tensorflow Mobile. Tensorflow Lite, yang dirilis pada November 14, 2017, merupakan versi perbaikan dari Tensorflow Mobile yang telah dirilis terlebih dahulu. Tensorflow Lite mendukung penggunaan CNN, DNN, dan RNN sebagai model Neural Network.

Untuk menjalankan inference, Tensorflow Lite menggunakan model Neural Network yang telah dilatih dan direpresentasikan dalam file dengan ekstensi .tflite. File ini sebenarnya merupakan graf komputasi yang berisi serangkaian node (operation) dan edge (tensor). Model file tersebut dimuat oleh C++ API untuk kemudian diteruskan ke interpreter. Interpreter mengeksekusi model menggunakan serangkaian kernel yang dimuat secara selektif (hanya kernel terkait saja yang dimuat). Gambar 1.7 merupakan ilustrasi dari arsitektur Tensorfow Lite.



Gambar 1.7: Ilustrasi dari arsitektur Tensorflow Lite.

Berbeda dengan Tensorflow Mobile, Tensorflow memiliki kernel tersendiri yang

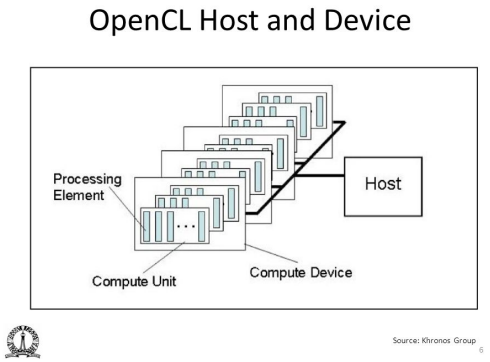
berisi operasi-operasi *Deep Learning inference* dan terpisah dari core Tensorflow. Sehingga, modifikasi kernel Tensorflow Lite dengan menambahkan program OpenCL atau Vulkan tidak akan mengubah perilaku dari program utama Tensorflow.

Dibandingkan dengan Tensorflow Mobile, Tensorflow Lite telah mendapatkan berbagai macam optimisasi.Performa inference yang dihasilkan lebih baik daripada Tensorflow Mobile. Beberapa bug pada Tensorflow Mobile juga telah diperbaiki pada Tensorlfow Lite. Selain itu, fitur-fitur yang ditawarkan Tensorflow Lite juga lebih banyak dibandingkan Tensorflow Mobile. Berikut adalah beberapa keunggulan Tensorflow Lite dibandingkan Tensorflow Mobile [21].

1. Memiliki fitur yang dapat memprune node dari operation graph yang tidak terpakai pada proses inference.
2. Memiliki kernel yang mendukung komputasi dalam 8-bit fixed point.
3. Optimasi performa untuk operasi floating point maupun fixed point.
4. Menggunakan pre-fused activation pada model Deep Learning.
5. Memuat kernel secara selektif sehingga menghemat memori.

1.10 OpenCL

OpenCL merupakan API untuk pemrograman paralel dengan memanfaatkan prosesor yang berbeda-beda seperti CPU, GPU, DSP, dan FPGA, sehingga dapat meningkatkan performa komputasi secara signifikan [22]. Pada OpenCL terdapat dua sisi program, yaitu host dan device. Device adalah prosesor target, sedangkan host adalah yang mengatur jalannya program pada device. Tugas yang dilakukan oleh host antara lain mencari device, menginisialisasi konteks, mengkompilasi kernel, menyalin data dari host ke device memory, menjalankan kernel, dan mengambil data hasil komputasi dari device memory. Program host ditulis dalam bahasa C++ dengan ekstensi OpenCL. Gambar 1.8 mengilustrasikan bagaimana host mengatur komputasi pada device.



Gambar 1.8: Host mengatur jalannya komputasi pada satu atau lebih device.

Program yang dijalankan pada device merupakan sekumpulan OpenCL kernel. Kernel ditulis menggunakan bahasa C dengan ekstensi OpenCL. Pada bagian host, kernel ditulis sebagai string dan dikompilasi terlebih dahulu sebelum dapat dieksekusi. Beberapa hal yang perlu dilakukan host sebelum menjalankan kernel pada OpenCL device adalah sebagai berikut.

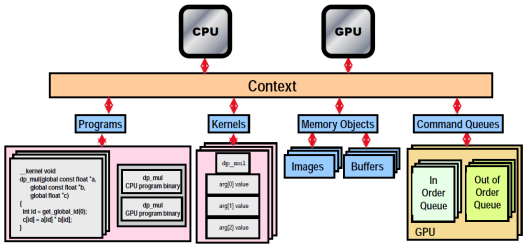
1. Membuat context. Context adalah environment dimana kernel dieksekusi. Pada context didefiniskan kernel yang digunakan, device yang digunakan, buffer/memory yang dapat diakses oleh device, properti dari memori tersebut, dan satu atau lebih command-queue untuk penjadwalan eksekusi kernel.

2. Mempersiapkan kernel. Kernel adalah serangkaian instruksi yang dijalankan pada device. Kernel ditulis sebagai string dan dikompilasi menggunakan fungsi clBuildKernel() dari OpenCL API.

3. Mempersiapkan buffer. Buffer object merupakan memory object yang menyimpan koleksi data secara linear dalam bytes. Buffer object dapat diakses menggunakan pointer pada kernel yang berjalan pada device. Konten dari buffer juga dapat dimanipulasi melalui host.

4. Membuat command-queue. Command-queue merupakan objek yang menampung perintah-perintah yang akan dieksekusi pada device. Suatu command-queue dibuat unutk device yang spesifik dalam sebuah konteks.

Kernel dapat disubmit ke command-queue dengan mendefinisikan banyaknya work-item dan work-group yang megeksekusi kernel. Gambar 1.9 mengilustrasikan bagaimana OpenCL kernel dieksekusi pada device.



Gambar 1.9: Cara kerja OpenCL.

OpenCL program dapat berjalan pada perangkat yang memiliki dukungan library OpenCL (libOpenCl.so). Pada NVidia GPU, library dapat diperoleh dalam paket installasi CUDA. Sedangkan pada perangkat android, beberapa vendor GPU seperti Adreno dan Mali menyertakan library OpenCL pada perangkat mereka. Library libOpenCl.so biasanya terletak pada direktori /system/vendor/lib/ dan dapat diambil melalui perintah adb pull pada Linux. Berikut adalah contoh program host dan device (kernel) pada OpenCL.

```
/* OpenCL Host Example */
void clHello() {
    cl_device_id device_id = NULL;
    cl_context context = NULL;
    cl_command_queue command_queue = NULL;
    cl_mem memobj = NULL;
    cl_program program = NULL;
    cl_kernel kernel = NULL;
    cl_platform_id platform_id = NULL;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;
    cl_int ret;

    char string[MEM_SIZE];

    /* Get Platform and Device Info */
    ret = clGetPlatformIDs(1, &platform_id,
    &ret_num_platforms);
    ret = clGetDeviceIDs( platform_id,
    CL_DEVICE_TYPE_DEFAULT, 1, &device_id,
    &ret_num_devices);

    /* Create OpenCL context */
```

```
context = clCreateContext( NULL, 1, &device_id,
NULL, NULL, &ret);

/* Create Command Queue */
command_queue = clCreateCommandQueue(context,
device_id, 0, &ret);

/* Create Memory Buffer */
memobj = clCreateBuffer(context,
CL_MEM_READ_WRITE,MEM_SIZE * sizeof(char), NULL,
&ret);

/* Create Kernel Program from the source */
program = clCreateProgramWithSource(context, 1,
(const char **)&CLCL_HELLO,
NULL, &ret);

/* Build Kernel Program */
ret = clBuildProgram(program, 1, &device_id, NULL,
NULL, NULL);

/* Create OpenCL Kernel */
kernel = clCreateKernel(program, "hello", &ret);

/* Set OpenCL Kernel Arguments */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),
(void *)&memobj);

/* Execute OpenCL Kernel */
ret = clEnqueueTask(command_queue, kernel, 0,
NULL,NULL);

/* Copy results from the memory buffer */
ret = clEnqueueReadBuffer(command_queue, memobj,
CL_TRUE, 0,
MEM_SIZE * sizeof(char),string, 0, NULL, NULL);
```

```
    /* Display Result */
    __android_log_print(ANDROID_LOG_DEBUG, "OpenCL",
    "Hello: %s", string);


    /* Finalization */
    ret = clFlush(command_queue);
    ret = clFinish(command_queue);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(memobj);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);
}
```

```
/* OpenCL kernel example */
const char *CLCL_HELLO =
    "__kernel void hello(__global char* string)\n"
    "{\n"
    "    string[0] = 'H';\n"
    "    string[1] = 'e';\n"
    "    string[2] = 'l';\n"
    "    string[3] = 'l';\n"
    "    string[4] = 'o';\n"
    "    string[5] = '\\0';\n"
    "}\n"
    "";
```

1.11 Vulkan

Vulkan merupakan API untuk menjalankan operasi-operasi grafik dan komputasi pada GPU. Vulkan merupakan generasi baru dari API pemrograman GPU pada android yang sebelumnya memanfaatkan OpenGL [23]. Vulkan memungkinkan akses multi-device dan menawarkan efisiensi yang tinggi untuk komputasi pada GPU. Vulkan memiliki driver yang sederhana dengan overhead CPU yang lebih kecil pada bagian host. Sama seperti OpenCL, Vulkan dirancang untuk dapat menjalankan program dengan multithreading.

Program pada Vulkan terdiri dari host dan device. Jika pada OpenCL instruksi yang dieksekusi pada device disebut kernel, pada Vulkan disebut shader. Selain untuk grafik, Vulkan juga dapat digunakan untuk melakukan komputasi sederhana sebagaimana yang dapat dilakukan pada OpenCL, misalnya perkalian matriks, penjumlahan vektor, atau dot product. Shader yang hanya melakukan operasi-operasi komputasi tersebut dinamakan Compute Shader. Pada Vulkan, compute shader dapat ditulis dalam GLSL (OpenGL Shading Language) yang selanjutnya dicompile menjadi SPIR-V binary.

GLSL merupakan bahasa high-level yang mirip dengan C/C++. Untuk komputasi sederhana seperti perkalian matriks atau konvolusi, shader terdiri dari layout dan main function. Main function merupakan fungsi yang mendefinisikan operasi utama. Layout digunakan untuk merepresentasikan data-data yang digunakan dalam komputasi. Data dibind pada host, kemudian layout juga definisikan binding pada bagian memory yang berisi data tersebut.

Pada OpenCL, kompilasi kernel dilakukan bersama dengan jalannya program dan pada umumnya memakan waktu yang cukup lama. Kompilasi merupakan salah satu operasi paling memakan waktu dari keseluruhan program OpenCL. Dalam hal ini, Vulkan memiliki keunggulan. Selain dapat melakukan kompilasi shader sebagai bagian dari program, Vulkan juga dapat menerima shader dalam bentuk SPIR-V binary. Shader dapat dikompile terlebih dahulu di luar program. Selain menghemat waktu, penggunaan binary shader dapat menjaga kerahasiaan shader pada kode sumber.

Driver yang sederhana pada Vulkan memiliki konsekuensi API yang lebih low-level. Untuk melakukan komputasi diperlukan persiapan yang lebih rumit pada bagian host jika dibandingkan dengan OpenCL. Beberapa hal yang perlu dilakukan antara lain sebagai berikut [23].

1. Membauat Vulkan Instance. Instance merupakan representasi dari Vulkan library yang menyimpan state komputasi. Instance pada Vulkan mirip seperti konteks pada OpenCL. Untuk membuat instance perlu diketahui informasi mengenai API Vulkan pada perangkat android yang digunakan, misalnya versi dari Vulkan.

2. Mencari Physical Device. Physical device adalah representasi dari fisik device (GPU) yang mendukung Vulkan. Masing-masing device mendukung jenis pekerjaan tertentu. Harus ditemukan device sesuai dengan jenis pekerjaan yang akan dilakukan. Jenis pekerjaan dapat berupa pemrosesan grafik atau komputasi. Physical device akan digunakan untuk membuat logical device.

3. Membuat logical device dan queue. Logical device merupakan perantara yang memungkinkan aplikasi untuk berinteraksi dengan device fisik. Pembuatan logical device disertai dengan membuatan queue. Setiap physical device memiliki beberapa jenis queue untuk jenis pekerjaan yang berbeda-beda, misalnya pemrosesan grafis atau komputasi biasa. Queue digunakan untuk menampung dan menjadwalkan perintah-perintah yang akan dieksekusi pada device. Pada logical device perlu diberikan informasi jenis queue yang digunakan.

4. Membuat Buffer. Sama seperti pada OpenCL, buffer merupakan tempat untuk meletakkan data-data yang diperlukan dalam eksekusi shader. Data-data tersebut disalin dari host. Buffer pada Vulkan tidak dapat mengalokasikan memory sendiri. Alokasi memori untuk buffer dilakukan secara manual oleh programmer. Seperti queue, terdapat beberapa jenis buffer untuk jenis pekerjaan yang berbeda-beda.

5. Membuat Descriptor Set. Descriptor set merupakan set yang berisi referensi-referensi terhadap resource yang digunakan pada eksekusi shader. Buffer merupakan salah satu contoh resource. Agar dapat digunakan oleh shader, descriptor set diikat ke pipeline. Pembuatan descriptor set didahului dengan pembuatan descriptor set layout. Descriptor set layout mendeskripsikan jenis, ukuran, dan urutan dari sumber daya pada descriptor set. Pada descriptor set layout didefinisikan binding dari setiap buffer object. Selain layout, perlu juga mengalokasikan descriptor pool yang menampung setiap descriptor set yang akan dibuat.

6. Membuat shader. Shader adalah perintah yang akan dijalankan di device. Terdapat beberapa jenis shader pada Vulkan, misalnya vertex shader dan compute shader. Compute shader digunakan untuk keperluan komputasi biasa, contohnya perkalian matriks. Shader dapat ditulis dalam bahasa GLSL (OpenGL Shading Language) yang mirip dengan C/C++. Shader dapat dikompilasi bersama dengan jalnnya program atau dapat langsung diimport sebagai SPIR-V binary. Shader yang telah dibungkus sebagai suatu objek (dinamakan shader module) akan diikat ke pipeline bersama dengan sumber daya yang diperlukan.

7. Membuat Compute Pipeline. Compute pipeline merupakan jalur yang digunakan oleh perintah-perintah komputasi untuk dapat dieksekusi oleh GPU. dalam membuat pipeline perlu didefinisikan shader yang digunakan beserta

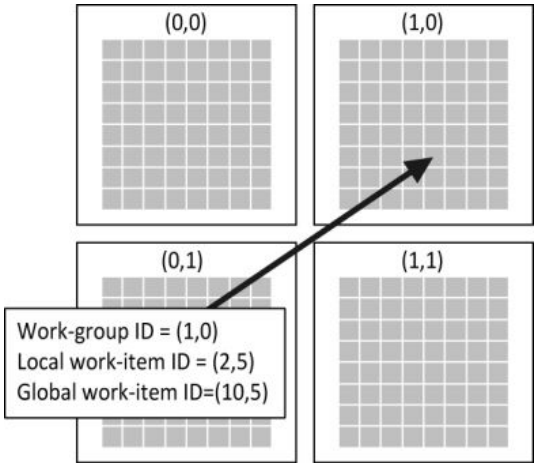
jenis dan fungsi utamanya. Selain itu juga perlu didefinisikan descriptor set yang digunakan.

8. Membuat Command Buffer. Command buffer digunakan untuk merekam semua perintah-perintah yang akan dikirimkan ke GPU. Untuk membuat command buffer perlu dialokasikan terlebih dahulu command pool yang menampung command buffer tersebut. Pada command pool perlu didefinisikan queue yang digunakan. Eksekusi shader dilakukan dengan perintah dispatch pada command buffer. Command buffer yang telah selesai merekam perintah dapat disubmit ke queue untuk dieksekusi oleh device.

1.12 SIMD pada OpenCL dan Vulkan

GPU merupakan device yang memiliki banyak unit komputasi (thread). OpenCL dan Vulkan mendukung pemrograman paralel pada level data. Instruksi untuk GPU dijalankan oleh banyak unit komputasi yang bekerja secara paralel. Masing-masing unit komputasi menjalakan instruksi yang sama namun pada bagian data yang berbeda-beda. Misalnya pada penjumlahan matriks, masing-masing unit pemrosesan hanya memproses elemen pada baris dan kolom yang sama saja. Dalam OpenCL dan Vulkan, unit komputasi ini disebut work item [22] [23]. Setiap work item mengeksekusi kernel atau shader yang sama. Beberapa work item dapat membentuk suatu local group (work group). Jika secara global work item bekerja secara independen, sinkronisasi komputasi dapat terjadi pada seluruh work item dalam satu work group.

Setiap work item memiliki identifier (ID) yang unik. ID adalah bilangan bulat positif yang dimulai dari 0. ID dari work-item terdiri dari dua jenis, yaitu global ID yang berlaku secara global dan local ID yang berlaku secara local dalam satu work gorup. Setiap work group juga memiliki ID yang unik. Setiap work item dapat mengakses beberapa jenis memori, antara lain private memory, local memory, dan global memory. Masing-masing private memory hanya bisa diakses oleh satu work item. Local memory merupakan shared memory pada level work group. Semua work item dalam satu work group dapat mengakses memory ini. Sedangkan global memory dapat diakses oleh semua work item yang terlibat dalam komputasi. Urutan kecepatan akses memory pada OpenCL dan Vulkan dari yang tercepat hingga yang terlambat adalah private memory, local memory, kemudian global memory. Gambar 1.10 merupakan ilustrasi dari work item dan work group pada OpenCL dan Vulkan.



Gambar 1.10: Work group dan work item pada OpenCL dan Vulkan .

Saat suatu proses berjalan pada device, dapat diambil ID dari work item atau work group yang menjalankan proses tersebut. Dengan demikian dapat diatur bagian data mana yang diproses oleh masing-masing ID. Misalnya dalam penjumlahan vektor, work item dengan ID i hanya memproses elemen ke i dari vektor-vektor yang dijumlahkan.

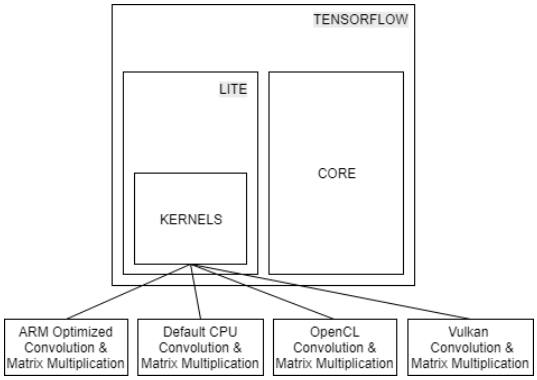
BAB 2
METODOLOGI

Pada bagian ini akan dijelaskan metodologi yang digunakan oleh Penulis pada saat melakukan penelitian. Metodologi terdiri dari metode implementasi OpenCL dan Vulkan serta metode pengujian hasil implementasi.

2.1 Metode Implementasi

Deep Learning Inference terdiri dari operasi-operasi matriks. Penerapan Deep Learning Inference menggunakan GPU dapat dilakukan dengan menerapkan OpenCL dan Vulkan pada operasi-operasi matriks terkait. Misalnya pada DNN inference, OpenCL dan Vulkan dapat diterapkan terhadap operasi perkalian matriks-vector untuk dijalankan pada GPU. Pada CNN inference, selain perkalian matriks juga terdapat operasi konvolusi yang dapat dijalankan melalui OpenCL dan Vulkan pada GPU. Dalam penelitian ini penulis hanya berfokus pada dua operasi, perkalian matriks dan convolution.

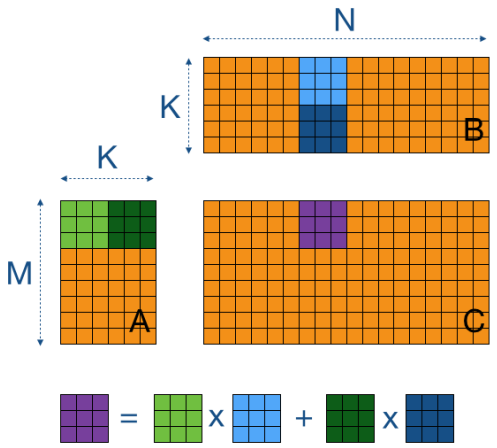
Implementasi Deep Learning Inference pada GPU dilakukan melalui Tensorflow Lite sebagai framework Deep Learning Inference pada perangkat mobile. Tensorflow Lite memiliki implementasi kernel berisi operasi-operasi Neural Network inference yang terpisah dari Tensorflow core. Penulis menambahkan dukungan OpenCL dan Vulkan terhadap kernel untuk operasi perkalian matriks dan konvolusi sehingga DNN dan CNN dapat dijalankan pada GPU. Tensorflow Lite tidak memiliki algoritma perkalian matrix-vector. Untuk DNN, ia menggunakan algoritma perkalian matrix biasa. Saat kompilasi, aplikasi dapat memilih untuk menggunakan default kernel (CPU), OpenCL kernel (GPU), atau Vulkan kernel (GPU). Gambar 2.1 menunjukkan struktur dari Tensorflow Lite dan bagian dimana OpenCL dan Vulkan diterapkan.



Gambar 2.1: Bagian Tensorflow Lite yang dioptimasi.

Dalam mengimplementasikan perkalian matriks menggunakan OpenCL dan Vulkan, penulis menggunakan pendekatan multiplication by block (blocking). Perkalian matriks dikerjakan secara bertahap dengan membagi matriks ke dalam beberapa blok. Pendekatan ini didasarkan pada paper [15]. Motivasi dari pendekatan blocking ini adalah memanfaatkan kemampuan memori lokal (dalam suatu workgroup) yang lebih cepat daripada memori global. Perkalian matriks konvensional akan selalu mengambil data dari buffer yang terletak di global memory. Diketahui bahwa perkalian matriks melibatkan pembacaan data yang sama beberapa kali. Misalnya pada perkalian matriks A dan B yang menghasilkan matriks C, untuk mendapatkan nilai baris ke-i pada matriks C dilakukan dot product antara baris ke-i dari matriks A dengan setiap kolom dari matriks B. Pada perkalian matriks biasa, baris ke-i dari matriks A akan dibaca beberapa kali melalui global memory.

Dengan melakukan perkalian matriks per-blok, keunggulan kecepatan akses local memori dapat dimanfaatkan. Pasangan blok matriks A dan B yang akan dikalikan dimuat terlebih dahulu ke local memory. Block tersebut hanya dibaca satu kali dari global memory selama proses komputasi. Ketika perkalian antar blok dilakukan, banyak data yang sama pada blok tersebut yang dibaca berkali-kali, namun kali ini dibaca melalui local memory. Dengan demikian akan lebih sedikit waktu yang terbuang untuk operasi read memory. Perhatikan bahwa setiap blok matriks masing-masing menggunakan data yang unik. Data pada blok-blok lain tidak dibaca ketika memproses suatu blok. Sehingga tidak ada data yang dibaca lebih dari satu kali dari global memory. Gambar 2.2 menujukkan contoh perkalian matriks pada dengan blocking.



Gambar 2.2: Perkalian matriks per blok.

Selain menggunakan block, beban komputasi dibagi ke banyak thread (work item) pada GPU. Satu thread hanya memproses perkalian untuk menghasilkan satu elemen pada matriks output. Jika matriks output merukuran MxN, maka akan terdapat MxN thread. Dengan demikian terjadi reduksi kompleksitas karena semua thread berjalan secara parallel. Kompleksitas perkalian matriks pada GPU dapat dihitung sebagai berikut. //Hitung kompleksitas

//Pendekatan implementasi matrix convolution Kemudian untuk mengimplementasikan konvolusi matriks juga digunakan pendekatan yang sama yaitu dengan memanfaatkan local memori untuk mempercepat proses pembacaan data. Pendekatan ini mengikuti paper [16].

Implementasi perkalian matriks dan konvolusi melalui OpenCL dan Vulkan memerlukan persiapan yang cukup panjang sebelum komputasi seperti yang dijelaskan pada bagian landasan teori OpenCL dan Vulkan. Oleh karena itu, penulis mengoptimisasi proses persiapan ini dengan memindahkan beberapa persiapan ke awal mulainya aplikasi sehingga hanya dijalankan satu kali selma aplikasi berjalan. Tidak semua persiapan dapat dipindahkan ke awal jalannya aplikasi karena beberapa persiapan bergantung pada data input yang bersifat dinamis, sehingga tidak dapat dilakukan hanya satu kali ketika aplikasi dijalankan.

// Pada OpenCL apa saja yang diawal, pada Vulkan apa saja.

2.2 Metode Pengujian

Setelah implementasi OpenCL dan Vulkan menghasilkan keluaran yang benar, dilakukan pengujian untuk mengetahui performa dari masing-masing implementasi. Kemudian, performa dibandingkan dengan implementasi asli dari Tensorflow serta implementasi multithreading yang sudah dioptimalkan untuk arsitektur ARM. Un-

tuk melakukan pengujian, penulis menggunakan aplikasi demo dari Tensorflow yaitu Tensorflow Lite Camera Demo yang merupakan aplikasi image recognition yang memanfaatkan CNN. Beberapa perangkat android dengan GPU Adreno dan Mali digunakan dalam pengujian ini. Selain melakukan pengujian pada perangkat android, penulis juga melakukan pengujian pada PC untuk mengetahui perbedaan performa pada PC dan android. Penulis melakukan pengujian terhadap beberapa model CNN yang mengandung convolution layer dan fully-connected layer, antara lain Inception, LeNet, dan AlexNet.

Dalam pengujian, penulis mengamati tiga hal yaitu kecepatan inference, penggunaan memori, dan penggunaan baterai. Selain itu, penulis juga mengamati kecepatan dari masing-masing operasi perkalian matriks dan konvolusi yang terlibat saat proses inference. Penulis kemudian juga mengamati kontribusi proses-proses persiapan OpenCL dan Vulkan terhadap kecepatan dari keseluruhan inference untuk mengetahui letak bottleneck pada inference, terutama pada matriks-matriks kecil. Penulis juga membandingkan performa eksekusi kernel/shader antara OpenCL dan Vulkan untuk mengetahui performa driver dari masing-masing API.

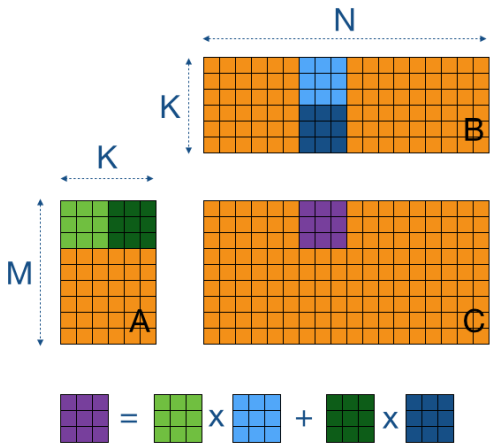
BAB 3
IMPLEMENTASI

Bagian ini berisi penjelasan mengenai implementasi program-program yang terkait dengan penelitian. Program tersebut meliputi perkalian matriks, kovolusi matriks, transpose matriks, serta penjumlahan matriks yang diimplementasikan menggunakan OpenCL dan diintegrasikan ke Tensorflow Lite.

3.1 OpenCL Matriks Multiplication

Ukuran blok yang digunakan disesuaikan dengan ukuran work-group yang disarankan pada OpenCL. Pada umunya perangkat android menyarankan ukuran kelipatan 8, sehingga ukuran blok dapat diambil 8x8, 16x32 atau 32x32. Ukuran lebih besar dari 32 tidak disarankan karena mayoritas perangkat memiliki batas jumlah work-item dalam suatu work-group sebanyak 1024 (32x32). ///////////

Implementasi perkalian matriks menggunakan blocking matrix multiplication. Kedua matriks yang dikalikan dibagi ke dalam beberapa block yang sama dimana satu block memiliki ukuran 32x32. Perhatikan pada gambar , terdapat dua matriks A dan B yang masing-masing berukuran MxK dan KxN. Dengan demikian perkalian keduanya menghasilkan matriks C yang berukuran MxN. Blok pada matriks C yang berwarna ungu diperoleh dari dua tahap perkalian matriks, yaitu perkalian antara blok hijau muda dan blok biru muda pada matriks A dan B kemudian ditambahkan dengan perkalian antara blok hijau tua dan blok biru tua pada matriks A dan B.



Gambar 3.1: Perkalian matriks per blok.

Motivasi dari penggunaan blok pada perkalian matriks adalah untuk meningkatkan efisiensi penggunaan cache pada GPU. Perhatikan bahwa untuk mendapatkan suatu elemen baris ke i dan kolom ke j dari matriks, baris ke i dari matriks A dan kolom ke j dari akan diload ke memory dan cache. Kemudian untuk mendapatkan suatu elemen baris ke i+1 dan kolom ke j dari matriks, kolom ke j dari matriks B kembali diload ke memory. Kolom ke j dari matriks B ini selalu digunakan ulang dalam menghitung elemen kolom ke j pada matriks C.

Untuk matriks berukuran besar, besar kemungkinan bahwa kolom tersebut sudah terhapus dari cache karena sebelum memproses kolom itu lagi, cache terlebih dahulu digunakan untuk menyimpan kolom-kolom lain karena sudah banyak perkalian yang dilakukan. Sedangkan ketika matriks berukuran kecil, peluang kolom tersebut masih berada di cache lebih tinggi, sehingga komputasi menggunakan kolom tersebut menjadi lebih cepat. Inilah mengapa matriks dibagi ke dalam beberapa bagian yang lebih kecil untuk dikalikan secara bertahap agar frekuensi penggunaan cache lebih tinggi sehingga komputasi lebih cepat.

Perkalian per blok ini dapat diimplementasikan dalam OpenCL kernel seperti berikut. Iterasi dilakukan sebanyak K/blocksize kali. Pada setiap iterasi, perkalian per blok dilakukan untuk 32 kolom dari matriks A (semua baris) dan 32 baris dari matriks B (semua kolom). Ingat bahwa ukuran blok adalah 32x32.

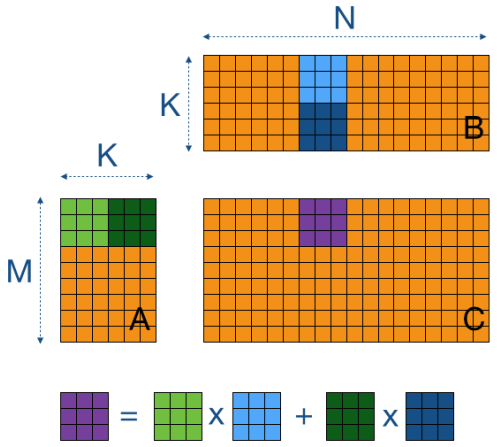
```
__kernel void matrixVectorMul(__global float* C,
    const __global float* A,
    const __global float* B,
    int K, int M, int N) {

        // Local row ID (max: 32)
    const int row = get_local_id(0);
    // Local col ID (max: 32)
    const int col = get_local_id(1);
    // Row ID of C (0..M)
    const int globalRow = 32*get_group_id(0) + row;
    // Col ID of C (0..N)
    const int globalCol = 32*get_group_id(1) + col;

    __local float Asub[32][32];
    __local float Bsub[32][32];

    float acc = 0.0;
```

```
    const int numTiles = ((K-1)/32)+1;
    for (int t=0; t<numTiles; t++) {

        const int tiledRow = 32*t + row;
        const int tiledCol = 32*t + col;
        if((tiledCol < K) && (globalRow < M)) {
            Asub[col][row] = A[globalRow*K + tiledCol];
        }
        else {
            Asub[col][row] = 0.0;
        }
        if((tiledRow < K) && (globalCol < N)) {
            Bsub[col][row] = B[globalCol*K + tiledRow];
        }
        else {
            Bsub[col][row] = 0.0;
        }

        barrier(CLK_LOCAL_MEM_FENCE);

        for (int k=0; k<32; k++) {
            acc += Asub[k][row] * Bsub[col][k];
        }

        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if((globalRow < M) && (globalCol < N)) {
        C[globalCol*M + globalRow] = acc;
    }
}
```

3.2 OpenCL Matriks Convolution

3.3 Vulkan Matriks Multiplication

Implementasi perkalian matriks menggunakan blocking matrix multiplication. Kedua matriks yang dikalikan dibagi ke dalam beberapa block yang sama dimana satu block memiliki ukuran 32x32. Perhatikan pada gambar , terdapat dua matriks A dan B yang masing-masing berukuran MxK dan KxN. Dengan demikian perkalian keduanya menghasilkan matriks C yang berukuran MxN. Blok pada matriks C yang berwarna ungu diperoleh dari dua tahap perkalian matriks, yaitu perkalian antara blok hijau muda dan blok biru muda pada matriks A dan B kemudian ditambahkan dengan perkalian antara blok hijau tua dan blok biru tua pada matriks A dan B.



Gambar 3.2: Perkalian matriks per blok.

Motivasi dari penggunaan blok pada perkalian matriks adalah untuk meningkatkan efisiensi penggunaan cache pada GPU. Perhatikan bahwa untuk mendapatkan suatu elemen baris ke i dan kolom ke j dari matriks, baris ke i dari matriks A dan kolom ke j dari akan diload ke memory dan cache. Kemudian untuk mendapatkan suatu elemen baris ke i+1 dan kolom ke j dari matriks, kolom ke j dari matriks B kembali diload ke memory. Kolom ke j dari matriks B ini selalu digunakan ulang dalam menghitung elemen kolom ke j pada matriks C.

Untuk matriks berukuran besar, besar kemungkinan bahwa kolom tersebut sudah terhapus dari cache karena sebelum memproses kolom itu lagi, cache terlebih dahulu digunakan untuk menyimpan kolom-kolom lain karena sudah banyak perkalian yang dilakukan. Sedangkan ketika matriks berukuran kecil, peluang kolom tersebut masih berada di cache lebih tinggi, sehingga komputasi menggunakan kolom tersebut menjadi lebih cepat. Inilah mengapa matriks dibagi ke dalam

beberapa bagian yang lebih kecil untuk dikalikan secara bertahap agar frekuensi penggunaan cache lebih tinggi sehingga komputasi lebih cepat.

Perkalian per blok ini dapat diimplementasikan dalam OpenCL kernel seperti berikut. Iterasi dilakukan sebanyak K/blocksize kali. Pada setiap iterasi, perkalian per blok dilakukan untuk 32 kolom dari matriks A (semua baris) dan 32 baris dari matriks B (semua kolom). Ingat bahwa ukuran blok adalah 32x32.

```
__kernel void matrixVectorMul(__global float* C,
const __global float* A,
const __global float* B,
int K, int M, int N) {

// Local row ID (max: 32)
const int row = get_local_id(0);
// Local col ID (max: 32)
const int col = get_local_id(1);
// Row ID of C (0..M)
const int globalRow = 32*get_group_id(0) + row;
// Col ID of C (0..N)
const int globalCol = 32*get_group_id(1) + col;

__local float Asub[32][32];
__local float Bsub[32][32];

float acc = 0.0;

const int numTiles = ((K-1)/32)+1;
for (int t=0; t<numTiles; t++) {

const int tiledRow = 32*t + row;
const int tiledCol = 32*t + col;
if((tiledCol < K) && (globalRow < M)) {
Asub[col][row] = A[globalRow*K + tiledCol];
}
else {
Asub[col][row] = 0.0;
}
if((tiledRow < K) && (globalCol < N)) {
```

```
Bsub[col][row] = B[globalCol*K + tiledRow];
}
else {
Bsub[col][row] = 0.0;
}

barrier(CLK_LOCAL_MEM_FENCE);

for (int k=0; k<32; k++) {
acc += Asub[k][row] * Bsub[col][k];
}

barrier(CLK_LOCAL_MEM_FENCE);
}

if((globalRow < M) && (globalCol < N)) {
C[globalCol*M + globalRow] = acc;
}
}
```

3.4 Vulkan Matriks Convolution

BAB 4
EKSPERIMEN DAN ANALISIS

Bagian ini bersisi pelaksanaan dan hasil eksperimen serta analisis dari hasil tersebut.

4.1 Mengubah Tampilan Teks

4.2 Memberikan Catatan

4.3 Menambah Isi Daftar Isi

4.4 Memasukan PDF

4.5 Membuat Perintah Baru

BAB 5
KESIMPULAN DAN SARAN

@todo
tambahkan kata-kata pengantar bab 6 disini

5.1 Kesimpulan

5.2 Saran

DAFTAR REFERENSI

[1] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436444. https://doi.org/10.1038/nature14539

[2] Szegedy, C., Wei Liu, Yangqing Jia, Sermanet, P., Reed, S., Anguelov, D., Rabinovich, A. (2015). Going deeper with convolutions. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE. https://doi.org/10.1109/cvpr.2015.7298594

[3] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 8490. https://doi.org/10.1145/3065386

[4] Liu, S., & Deng, W. (2015). Very deep convolutional neural network based image classification using small training sample size. In 2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR). IEEE. https://doi.org/10.1109/acpr.2015.7486599

[5] Russakovsky, O., Deng, J., Su, H. et al. Int J Comput Vis (2015) 115: 211. https://doi.org/10.1007/s11263-015-0816-y

[6] O'Shea, Keiron & Nash, Ryan. (2015). An Introduction to Convolutional Neural Networks. ArXiv e-prints.

[7] Martn Abadi, dkk.. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[8] Saving and Restoring — TensorFlow. (n.d.). Retrieved December 19, 2017, from https://www.tensorflow.org/programmers_guide/saved_model

[9] A Tool Developer's Guide to TensorFlow Model Files — TensorFlow. (n.d.). Retrieved December 19, 2017, from https://www.tensorflow.org/extend/tool_developers/#freezing

[10] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16), Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. JMLR.org 2849-2858.

[11] Chandra, V., Lai, L., & Suda, N. (2017). Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activations. CoRR, abs/1703.03073.

[12] Latifi Oskouei, S. S., Golestani, H., Hashemi, M., & Ghiasi, S. (2016). CNNdroid. In Proceedings of the 2016 ACM on Multimedia Conference - MM 16. ACM Press. https://doi.org/10.1145/2964284.2973801

[13] B. (2017, November 29). BVLC/caffe. Retrieved December 20, 2017, from https://github.com/BVLC/caffe

[14] Huang, S., Xiao, S., & Feng, W. (2009). On the energy efficiency of graphics processing units for scientific computing. In 2009 IEEE International Symposium on Parallel & Distributed Processing. IEEE. https://doi.org/10.1109/ipdps.2009.5160980

[15] Matsumoto, K., Nakasato, N., & Sedukhin, S. G. (2012). Performance Tuning of Matrix Multiplication in OpenCL on Different GPUs and CPUs. 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. doi:10.1109/sc.companion.2012.59

[16] Nugteren, C., & Codreanu, V. (2015). CLTune: A Generic Auto-Tuner for OpenCL Kernels. 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip. doi:10.1109/mcsoc.2015.10

[17] Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015.

[18] Sanders, J., & Kandrot, E. (2010). CUDA by example: An introduction to general-purpose GPU programming. Upper Saddle River: Addison-Wesley.

[19] Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444. doi:10.1038/nature14539

[20] Nvidia White Paper. 2015. GPU-Based Deep Learning Inference: A Performance and Power Analysis

[21] https://www.tensorflow.org/mobile/tflite/

[22] Munshi, A. (2009). The OpenCL specification. 2009 IEEE Hot Chips 21 Symposium (HCS). doi:10.1109/hotchips.2009.7478342

[23] Khronos Group. (2018). Vulkan: A Specification (with all registered Vulkan exstension).

LAMPIRAN

LAMPIRAN 1