**UNIVERSITAS INDONESIA**

# DEEP LEARNING INFERENCE PADA MOBILE GPU DENGAN 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 MOBILE GPU DENGAN 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 Mobile GPU dengan 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 | : | Deep Learning Inference pada Mobile GPU dengan OpenCL dan Vulkan |

**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 Mobile GPU dengan 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.

Tugas Akhir ini disusun dalam waktu yang cukup singkat sehingga masih terdapat banyak kekurangan baik pada konten penelitian maupun pada struktur penulisan. Penulis mengharapkan kritik dan saran dari pembaca sebagai bahan pembelajaran bagi penulis untuk menyusun karya-karya lain 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. Pak Risman Adnan sebagai Pembimbing II,
4. Ryorda Triaptahadi sebagai rekan penelitian, dan
5. Teman-teman 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 kepada banyak 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 Mobile GPU dengan 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 Mobile GPU dengan OpenCL dan Vulkan |



*Deep Learning inference* saat ini sudah dapat dijalankan pada perangkat *mobile*. Mayoritas *library* untuk *mobile Deep Learning* hanya memungkinkan operasi-operasi matriks pada *inference* dijalankan menggunakan CPU. Penulis meneliti penggunaan GPU untuk menjalankan operasi-operasi matriks pada *mobile Deep Learning inference*. Penulis telah mengimplementasikan beberapa *kernel* tambahan pada Tensorflow Lite untuk operasi perkalian matriks-matriks dan konvolusi matriks yang berjalan di *mobile* GPU dan dapat digunakan oleh *convolution layer* dan *fully-connected layer* ketika *inference*. Penulis menggunakan OpenCL, API pemrograman paralel untuk berbagai jenis prosesor, untuk mengimplementasikan *kernel* tersebut. Pengujian yang dilakukan menunjukkan bahwa Tensorflow Lite OpenCL *kernels* untuk operasi perkalian matriks-matriks memiliki performa yang mampu mengungguli performa *naive kernel* dan *optimized kernel* dari Tensorflow Lite yang berjalan di CPU. Sementara itu OpenCL *kernel* untuk operasi konvolusi matriks mampu mengungguli performa *naive kernel* dan *optimized kernel* ketika kanal matriks masukan dan keluaran berukuran kecil.



Kata Kunci:
Deep Learning; GPU; OpenCL;

# ABSTRACT


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



*Deep Learning inference* saat ini sudah dapat dijalankan pada perangkat *mobile*. Mayoritas *library* untuk *mobile Deep Learning* hanya memungkinkan operasi-operasi matriks pada *inference* dijalankan menggunakan CPU. Penulis meneliti penggunaan GPU untuk menjalankan operasi-operasi matriks pada *mobile Deep Learning inference*. Penulis telah mengimplementasikan beberapa *kernel* tambahan pada Tensorflow Lite untuk operasi perkalian matriks-matriks dan konvolusi matriks yang berjalan di *mobile* GPU dan dapat digunakan oleh *convolution layer* dan *fully-connected layer* ketika *inference*. Penulis menggunakan OpenCL, API pemrograman paralel untuk berbagai jenis prosesor, untuk mengimplementasikan *kernel* tersebut. Pengujian yang dilakukan menunjukkan bahwa Tensorflow Lite OpenCL *kernels* untuk operasi perkalian matriks-matriks memiliki performa yang mampu mengungguli performa *naive kernel* dan *optimized kernel* dari Tensorflow Lite yang berjalan di CPU. Sementara itu OpenCL *kernel* untuk operasi konvolusi matriks mampu mengungguli performa *naive kernel* dan *optimized kernel* ketika kanal matriks masukan dan keluaran berukuran kecil.



Keywords:
Deep Learning; GPU; OpenCL;

# DAFTAR ISI

# DAFTAR GAMBAR

# DAFTAR TABEL

# BAB 1
# PENDAHULUAN

Karya tulis yang berjudul ”Deep Learning Inference pada Mobile GPU dengan OpenCL dan Vulkan” ini didahului dengan pembahasan mengenai latar belakang penelitian, permasalahan yang ingin diselesaikan, dan tujuan dari penelitian.

## 1.1 Latar Belakang

*Deep Learning* merupakan algoritma *Machine Learning* yang diketahui memiliki akurasi tinggi. *Deep Learning* memanfaatkan arsitektur *Neural Network* yang telah dilatih menggunakan sekumpulan data untuk melakukan prediksi skor atau label dari suatu data baru. Popularitas *Deep Learning* meningkat pesat dalam beberapa tahun terakhir. Salah satu jenis arsitektur *Deep Learning*, *Convolutional Neural Network* (CNN), memiliki peran besar dalam meningkatnya popularitas *Deep Learning*. CNN biasa digunakan dalam aplikasi pengenalan citra. Selain CNN, dalam *Deep Learning* juga terdapat jenis model lain seperti *Long Short-Term Memory* (LSTM) yang biasa digunakan untuk menyelesaikan permasalahan di bidang pengolahan bahasa.

Seiring meningkatnya popularitas *Deep Learning*, dukungan terus bermunculan dari berbagai pihak terutama yang terkait dengan peningkatan performa *Deep Learning*. Saat ini CPU tidak lagi menjadi pilihan utama untuk menjalankan *Deep Learning* pada komputer personal atau *server*. GPU telah diketahui dapat menjalankan *Deep Learning* dengan performa komputasi yang jauh lebih baik daripada CPU, terutama ketika melakukan *training*. Selain dalam hal performa, dukungan untuk mobilitas Deep Learning juga mulai bermunculan. Belum lama ini, Google merilis *open-source library* bernama Tensorflow Lite [3] yang memungkinkan pengguna menjalankan *Deep Learning inference* pada perangkat *mobile* dengan performa tinggi.

Saat ini operasi-operasi *Deep Learning inference* pada Tensorflow Lite hanya dapat dijalankan di CPU. Meskipun demikian performa yang dihasilkan sudah sangat baik. Pada Tugas Akhir ini penulis melakukan penelitian mengenai penggunaan GPU untuk menjalankan *Deep Learning inference* pada perangkat *mobile* melalui Tensorflow Lite. Penulis mengimplementasikan beberapa Tensorflow Lite *kernel* untuk beberapa operasi matriks pada *inference* yang berjalan di GPU. Men-

jalankan *Deep Learning inference* pada *mobile* GPU bukanlah hal yang baru. Sudah ada *library* bernama CNNdroid [5] yang dapat digunakan untuk menjalankan CNN *inference* pada *mobile* GPU. CNNdroid memanfaatkan Renderscript untuk menjalankan komputasi pada GPU.

Dalam penelitian ini penulis menggunakan OpenCL untuk menjalankan *Deep Learning inference* pada *mobile* GPU. OpenCL [10] merupakan API pemrograman paralel untuk berbagai jenis prosesor seperti CPU, GPU, dan FPGA. OpenCL merupakan API pemrograman paralel yang mendukung komputasi dengan paradigma SIMT (*Single Instruction Multiple Thread*). *Deep Learning inference* terdiri dari operasi-operasi matriks yang berpotensi untuk diimplementasikan secara SIMT pada GPU yang memiliki sangat banyak *compute unit* (*thread*). Implementasi *Deep Learning inference* pada *mobile* GPU melalui OpenCL diharapkan memberikan performa yang baik.

## 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 akan dijawab dalam penelitian ini.

1. Apakah OpenCL dapat digunakan untuk menjalankan operasi-operasi *inference* di GPU melalui Tensorflow Lite?

2. Bagaimana perbandingan kecepatan Tensorflow Lite *kernel* yang berjalan di GPU melalui OpenCL dengan Tensorflow Lite *kernel* yang berjalan di CPU?

3. Dimanakah letak *bottleneck* (pada komputasi atau pada baca/tulis memori) dari penggunaan OpenCL untuk menjalankan operasi-operasi *inference* di GPU?

### 1.2.2 Batasan Permasalahan

Berikut adalah batasan dan asumsi pada penelitian.

1. Implementasi Tensorflow Lite *kernel* yang menggunakan OpenCL hanya dapat digunakan untuk perangkat *Android* saja.

2. Penulis hanya mengimplementasikan Tensorflow Lite *kernel* melalui OpenCL untuk dua operasi *inference* saja yaitu perkalian konvolusi matriks dan perkalian matriks-matriks.

3. Implementasi konvolusi matriks hanya dapat digunakan untuk konvolusi dengan *stride* sebesar 1 ke kanan dan 1 ke bawah.

4. Hasil implementasi OpenCL hanya dapat digunakan untuk *convolution layer* dan *fully-connected layer* saja.

5. 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 operasi-operasi *Deep Learning inference* berupa Tensorflow Lite *kernel* menggunakan OpenCL agar dapat dijalankan di GPU.

2. Membandingkan kecepatan Tensorflow Lite *kernel inference* yang berjalan di GPU melalui OpenCL dengan Tensorflow Lite *kernel* yang berjalan di CPU.

3. Mengetahui letak *bottleneck* (pada komputasi atau pada baca/tulis memori) dari penggunaan OpenCL untuk menjalankan operasi-operasi *inference* di GPU.

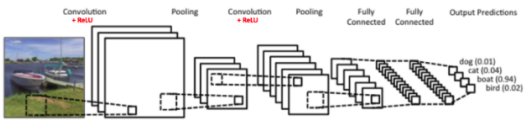
# BAB 2
# LANDASAN TEORI

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

## 2.1 Deep Learning

*Deep Learning* merupakan salah satu bentuk *Representation Learning* yang memungkinkan suatu mesin untuk diberikan sekumpulan data mentah, kemudian mesin tersebut dapat secara otomatis menemukan representasi data (*feature vector*) yang diperlukan untuk melakukan klasifikasi atau deteksi. *Deep Learning* dikenal memiliki akurasi yang lebih tinggi daripada algoritma *Machine Learning* konvensional dalam melakukan klasifikasi atau deteksi. *Deep Learning* memanfaatkan arsitektur *Neural Network* sebagai model. Arsitektur ini terinspirasi dari otak manusia yang terdiri dari banyak neuron. Model *Deep Learning* dilatih menggunakan sekumpulan data yang telah diketahui labelnya. Oleh karena itu, *Deep Learning* merupakan salah satu bentuk dari *Supervised Learning* [6].

Contoh model *Deep Learning* yang sangat terkenal adalah *Convolutional Neural Network* (CNN). CNN didesain untuk memproses data berupa *multi-dimensional array*, contohnya data citra. CNN sering digunakan dalam aplikasi pengenalan citra dan deteksi objek pada citra. CNN tersusun dari beberapa *convolution layer* yang berfungsi untuk mengekstrak fitur-fitur dari data dengan cara melakukan konvolusi menggunakan suatu *filter* [9]. *Convolution layer* biasanya disambungkan ke *pooling layer*, yaitu *layer* yang berfungsi untuk mereduksi ukuran matriks dengan cara melakukan *downsampling*. Pada bagian akhir model biasanya juga terdapat *fully-connected layer* yang bertugas melakukan klasifikasi menggunakan fitur-fitur yang didapat dari proses konvolusi pada *convolution layer*. Gambar 2.1 merupakan contoh model CNN.

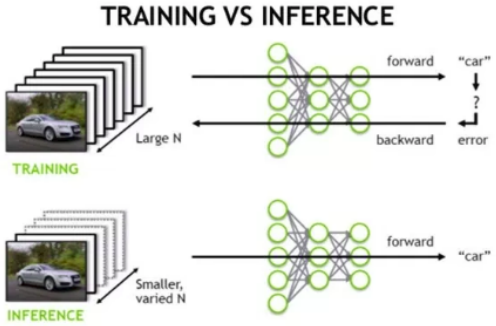



**Gambar 2.1:** Contoh model Convolutional Neural Network (CNN). Pada model ini terdapat *convolution layer*, *pooling layer*, dan *fully-connected layer*.

## 2.2 Deep Learning Training dan Inference

*Deep Learning* merupakan algoritma yang terdiri dari dua tahap, yaitu *training* dan *inference*. Pada tahap *training*, model dilatih menggunakan sekumpulan data. Pada awal *training*, *weight* inisial diberikan kepada model, biasanya berupa angka-angka acak berdasarkan suatu distribusi tertentu. Selanjutnya, data training dimasukkan ke model. Model melakukan *forward pass* atau prediksi terhadap data tersebut menggunakan *weight* sehingga akan diperoleh skor atau label dari data pada *output layer*. Label hasil prediksi kemudian dibandingkan dengan label asli dari data dan dihitung nilai galatnya menggunakan fungsi tertentu. Informasi galat ini kemudian dikirim kembali ke semua *layer* pada model dan digunakan untuk memperbarui nilai *weight* pada tiap *layer*. Proses mengirim kembali informasi galat ini disebut *back propagation* [11].

Model yang telah dilatih pada tahap *training* dapat digunakan untuk melakukan *inference*. *Inference* merupakan tahap dimana model melakukan *forward pass* terhadap data baru menggunakan model yang telah dilatih. Perbedaan utama dari *training* dan *inference* adalah bahwa pada *inference* tidak terdapat *back-propagation* setelah melakukan *forward-pass*, karena tujuan dari *inference* hanyalah mendapatkan hasil prediksi skor atau label dari data baru [11]. Gambar 2.2 menunjukkan perbedaan proses *training* dan *inference*. Terlihat bahwa pada *training* terjadi pemrosesan dua arah, sedangkan pada *inference* hanya satu arah.




**Gambar 2.2:** Perbedaan *training* dan *inference* pada *Deep Learning*. *Training* merupakan proses dua arah, sedangkan *inference* hanya satu arah.

## 2.3 Operasi-operasi Deep Learning Inference

Tahap *inference* pada *Deep Learning* melibatakan banyak operasi-operasi matriks. Konvolusi matriks dan perkalian matriks-matriks merupakan beberapa operasi pada *Deep Learning inference* yang memiliki beban komputasi besar. Dua operasi ini merupakan dua operasi yang diamati pada penelitian ini. Berikut adalah penjelasan

lebih lanjut mengenai tiga operasi tersebut.

### 2.3.1 Konvolusi Matriks

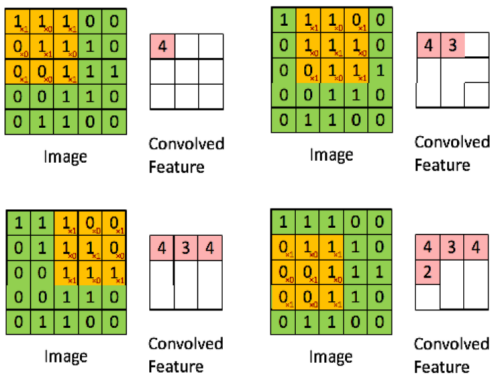
Operasi konvolusi matriks terjadi pada *convolution layer* dari CNN. Operasi ini melibatkan tiga buah matriks yaitu matriks *image*, matriks *filter*, dan matriks *output*. Matriks-matriks tersebut memiliki tiga dimensi yang merepresentasikan kanal, baris, dan kolom. Kanal adalah sumbu-z dari matriks. Banyaknya kanal menyatakan kedalaman matriks. Baris merupakan sumbu-y dari matriks. Banyaknya baris menyatakan ketinggian matriks. Kolom merupakan sumbu-x dari matriks. Banyaknya kolom menyatakan lebar matriks.

Pada *convolution layer*, *filter* dikonvolusikan terhadap matriks *image*. Suatu *image* dapat dikonvolusi menggunakan satu atau lebih *filter*. Karena itu, *filter* sebenarnya adalah matriks empat dimensi yaitu kanal, baris, kolom, dan *batch*. Besarnya *batch* menyatakan banyaknya *filter* tiga dimensi kanal, baris, dan kolom. Konvolusi dari satu *batch* dari *filter* menghasilkan satu kanal tersendiri pada matriks *output*, sehingga besarnya *batch* dari *filter* akan selalu sama dengan banyaknya kanal (kedalaman) *output* [9]. Berikut adalah beberapa aturan mengenai ukuran matriks *image*, *filter*, *bias*, dan *output* pada convolution layer.

1. Tinggi dan lebar *filter* selalu lebih kecil atau sama dengan tinggi dan lebar *image*.
2. Kedalaman *image* selalu sama dengan kedalaman *filter*.
3. Besarnya *batch* selalu sama dengan kedalaman *output*.
4. Tinggi dan lebar *bias* selalu sama dengan satu.
5. Kedalaman *bias* selalu sama dengan kedalaman *output*.
6. Tinggi dan lebar *output* selalu lebih kecil atau sama dengan tinggi dan lebar *image*.

Pada operasi konvolusi, posisi *filter* dimulai dari ujung kiri atas dari *image*. *Filter* kemudian bergeser ke kanan dan kebawah dengan jarak geser (*stride*) yang ditentukan (pada umumnya *stride* sama dengan satu). Pada suatu posisi *filter*, dilakukan *dot-product* antara *filter* dengan sub-matriks dari *image* yang bersesuaian dengan posisi filter. Satu kali operasi *dot-product* tersebut menghasilkan skalar yang merupakan satu elemen dari matriks *output* setelah ditambahkan bias dan diaktivasi. Setelah proses konvolusi selesai untuk satu *filter*, akan terbentuk satu kanal

*output*. Gambar 2.3 menunjukkan contoh operasi konvolusi dengan *image* berukuran $5 \times 5 \times 1$, *filter* berukuran $3 \times 3 \times 1$, dan nilai *stride* satu.

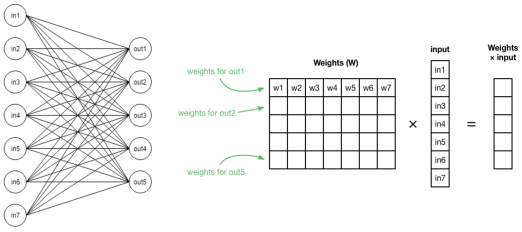



**Gambar 2.3:** Contoh operasi konvolusi. *Convolved feature* adalah matriks keluaran dari konvolusi.

Operasi konvolusi seperti di atas dapat dilakukan sekaligus untuk beberapa *batch*. Jika *batch* lebih dari satu, maka matriks-matriks *image* dan *output* adalah matriks empat dimensi yaitu kanal, baris, kolom, dan *batch*.

### 2.3.2 Perkalian Matriks-Matriks

Operasi perkalian matriks-matriks pada *Deep Learning inference* terjadi pada *fully-connected layer*. Pada *layer* ke-*l*, nilai *weight* disimpan dalam bentuk matriks dua dimensi berukuran $M \times K$, dimana $M$ adalah banyaknya *node* pada *layer* ke-*l* dan $K$ adalah banyaknya *node* pada *layer* ke-*(l-1)* [9]. Baris ke-*i* dari matriks *weight* pada *layer* ke-*l* tersebut merupakan vektor *weight* untuk *node* ke-*i* pada *layer* ke-*l*. Untuk memperoleh nilai semua *node* pada *layer* ke-*l*, matriks *weight* dikalikan dengan vektor sepanjang $K$ yang elemen-elemennya adalah nilai-nilai *node* pada *layer* ke-*(l-1)*. Hasilnya adalah vektor sepanjang $M$, sesuai banyaknya *node* pada *layer* ke-*l*. Ini dapat dilihat pada Gambar 2.4.




**Gambar 2.4:** Perkalian matriks-vektor pada *fully-connected layer*. Elemen ke-*i* pada vektor *weightxinput* adalah nilai *outi*.

Operasi perkalian matriks $M \times K$ dengan vektor $K \times 1$ pada *fully-connected layer* tersebut dapat dilakukan sekaligus untuk $N$ *batch*, sehingga akan terjadi op-

erasi perkalian matriks $M \times K$ dengan matriks $K \times N$ yang menghasilkan matriks $M \times N$.

Selain *fully-connected layer*, operasi perkalian matriks-matriks juga dapat terjadi pada *convolution layer*. Terdapat dua kasus yang menyebabkan terjadinya operasi ini. Kasus pertama adalah ketika operasi konvolusi melibatkan *filter* yang berukuran tinggi dan lebar $1 \times 1$. *Image* dapat dipandang sebagai matriks dua dimensi dengan tinggi $H_i \times W_i \times B_i$ dan lebar $C_i$ dimana $H_i$, $W_i$, $C_i$, dan $B_i$ berturut-turut adalah tinggi, lebar, banyaknya kanal, dan banyaknya *batch* dari *image*. Sementara itu *filter* dapat dipandang sebagai matriks dua dimensi dengan tinggi $C_f$ dan lebar $H_f \times W_f \times B_f$ dimana $H_f$, $W_f$, $C_f$, dan $B_f$ berturut-turut adalah tinggi, lebar, banyaknya kanal, dan banyaknya *batch* dari *filter*.

Kasus kedua terjadinya perkalian matriks-matriks pada *convolution layer* adalah ketika operasi konvolusi melibatkan *filter* yang tinggi dan lebarnya sama dengan tinggi dan lebar *image*. *Image* dapat dipandang sebagai matriks dua dimensi dengan tinggi $B_i$ dan lebar $H_i \times W_i \times C_i$, sedangkan *filter* dapat dipandang sebagai matriks dua dimensi dengan tinggi $H_f \times W_f \times C_f$ dan lebar $B_f$.

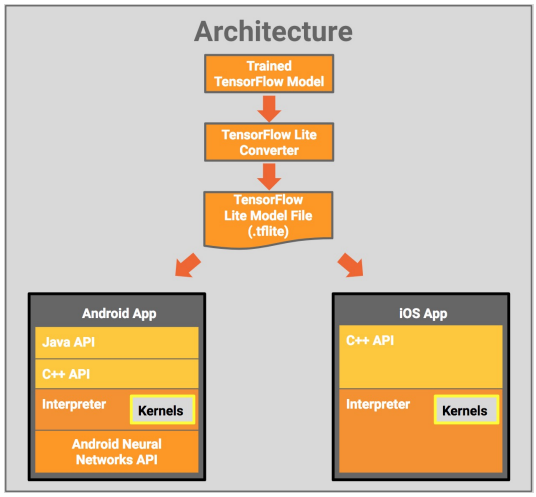
## 2.4 Tensorflow Lite

TensorFlow [8] adalah perangkat lunak *open-source* yang merupakan *library* untuk melakukan komputasi numerik menggunakan graf. *Node* pada graf merepresentasikan operasi matematika, sedangkan *edge* pada graf merepresentasikan *multi-dimensional data array* (*tensor*) sebagai masukan atau keluaran dari operasi pada *node*. TensorFlow pada awalnya dikembangkan oleh para pengembang dan peneliti dari Google untuk keperluan penelitian di bidang *Machine Learning* and *Deep Learning*. Saat ini Tensorflow merupakan salah satu *Machine Learning library* paling populer. API Tensorflow tersedia dalam bahasa C++, Python, Java, dan Javascript. Selain digunakan pada komputer personal atau *server*, Tensorflow juga dapat digunakan untuk menjalankan *Deep Learning inference* pada perangkat *mobile* melalui Tensorflow Mobile dan Tensorflow Lite.

Tensorflow Lite merupakan *Machine Learning library* terbaru dari Tensorflow yang ditujukan untuk perangkat Android dan IoS [3]. Saat ini Tensorflow Lite dapat memproses tiga jenis model *Deep Learning* yaitu CNN, DNN, dan LSTM. Berbeda dengan Tensorflow Mobile, Tensorflow Lite memiliki *kernel* tersendiri yang terpisah dari *core* Tensorflow. *Kernel* adalah implementasi operasi-operasi *Deep Learning infernce* seperti perkalian matriks, konvolusi, dan transpose matriks. Tensorflow Lite memiliki suatu jenis *kernel* yang telah dioptimalkan untuk

perangkat *mobile* (*optimized kernel*) yang memiliki performa yang sangat baik. Selain *optimized kernel*, terdapat pula *kernel* biasa (*naive kernel*) pada Tensorflow Lite. Namun tentu saja performanya tidak lebih baik dari *optimized kernel*.

Untuk melakukan *inference*, Tensorflow Lite menerima masukan model dengan format ".tflite". API akan meneruskan model masukan ke *interpreter* untuk diinterpretasi. *Interpreter* akan memuat semua *kernel* yang diperlukan oleh model tersebut secara selektif. Gambar 2.5 menjelaskan arsitektur Tensorfow Lite.




**Gambar 2.5:** Arsitektur Tensorflow Lite. *Interpreter* bertugas menginterpretasikan model ".tflite" dan memuat *kernels* yang diperlukan. *Kernels* tersebut terpisah dari *core* Tensorflow.

## 2.5 OpenCL

OpenCL merupakan API untuk melakukan pemrograman paralel pada prosesor yang berbeda-beda seperti CPU, GPU, DSP, dan FPGA [10]. OpenCL dapat digunakan untuk meningkatkan performa komputasi secara signifikan. OpenCL API menggunakan bahasa C/C++ dengan ekstensi. Pada OpenCL terdapat dua sisi program, yaitu *host* dan *device*. *Device* adalah prosesor target, tempat berjalannya komputasi. Sementara itu *host* adalah yang mengatur jalannya komputasi pada *device*. Menyalin data matriks dari *host memory* ke *device memory* dan menentukan banyaknya unit komputasi yang bekerja adalah contoh tugas dari *host*. Dalam penelitian ini program *host* berjalan pada CPU. *Device* pada penelitian ini adalah GPU. Dalam suatu program OpenCL terdapat istilah-istilah yang perlu dipahami sebagai berikut.

1. ***Context***. *Context* adalah lingkungan dimana komputasi pada *device* akan dilakukan. Pada *context* didefiniskan OpenCL *kernel* yang digunakan, *device* yang digunakan, memori (*buffer*) yang dapat diakses, properti dari memori

tersebut, dan satu atau lebih *command queue* untuk penjadwalan eksekusi OpenCL *kernel*.

2. ***Kernel***. OpenCL *kernel* merupakan serangkaian instruksi yang mendefinisikan suatu fungsi tertentu, contohnya perkalian matriks atau penjumlahan matriks, yang dieksekusi pada *device*. OpenCL *kernel* dikompilasi dan dijadwalkan eksekusinya oleh *host*. OpenCL *kernel* pada bagian *host* direpresentasikan dalam format *string* dan dikompilasi menggunakan fungsi $clBuildKernel()$ pada OpenCL API.

3. ***Buffer***. *Buffer* atau *buffer object* merupakan *memory object* yang menyimpan koleksi data secara linear dalam *bytes*. *Buffer* berada pada *device memory* (dalam penelitian ini adalah memori GPU). OpenCL *kernel* dapat mengakses data pada *buffer* menggunakan *pointer* yang diberikan melalui argumen *kernel*. Data pada *buffer* juga dapat dimanipulasi oleh *host*.

4. ***Command Queue***. *Command queue* merupakan objek yang menampung perintah-perintah yang akan dieksekusi pada *device*.

Tahap-tahap berjalannya suatu program OpenCL dapat dilihat pada Tabel 2.1. Sebelum suatu OpenCL *kernel* dapat dieksekusi pada *device*, *host* perlu melakukan beberapa persiapan seperti pada Tabel 2.1. Ketika persiapan telah selesai, OpenCL *kernel* dapat dijadwalkan eksekusinya dengan cara melakukan *enqueue* terhadap *command queue*. Perhatikan bahwa data-data yang terkait dengan eksekusi OpenCL *kernel* (data masukan dan data keluaran) perlu disalin secara manual dari *host memory* (CPU) ke *device memory* (GPU) dan sebaliknya.

**Tabel 2.1:** Tahap-tahap eksekusi program OpenCL dari awal hingga akhir.

| **No.** | **Tahap** |
|---|---|
| 1 | Membuat *context* |
| 2 | Membuat *kernel* |
| 3 | Membuat *comand-queue* |
| 4 | Membuat *buffer* |
| 5 | Menyalin data masukan dari *host memory* ke *device memory*. |
| 6 | Mendefinisikan struktur *work-space*. |
| 7 | Mendefinisikan argumen-argumen *kernel*. |
| 8 | Eksekusi *kernel*. |
| 9 | Menyalin data keluaran dari *device memory* ke *host memory*. |

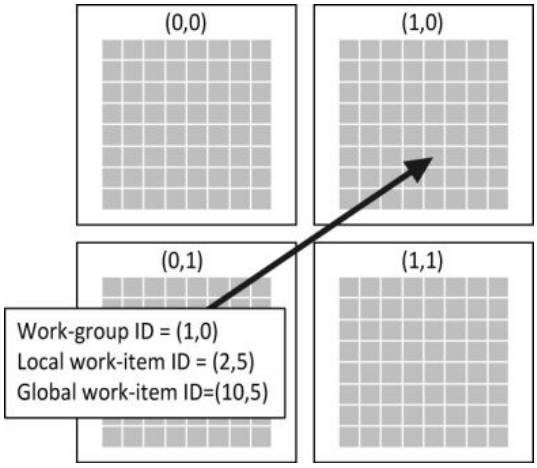
Untuk mengimplementasikan OpenCL, diperlukan OpenCL *library* (”libOpenCL.so”) dan OpenCL *headers* (”cl.h” dan ”cl_platform.h”). Pada NVIDIA GPU, OpenCL *library* dapat diperoleh dalam paket instalasi CUDA. Pada perangkat Android, OpenCL *library* disediakan oleh vendor dari masing-masing prosesor. Perangkat Android dengan vendor GPU Adreno atau Mali biasanya menyertakan OpenCL *library* yang terletak pada direktori ”/system/vendor/lib/”. *Library* tersebut dapat diambil menggunakan perintah ”adb pull” pada Linux.

## 2.6 SIMT pada OpenCL

GPU merupakan prosesor yang memiliki banyak unit komputasi (*thread*). Melalui OpenCL, komputasi pada GPU dapat dijalankan oleh banyak *thread* yang bekerja secara paralel, sehinnga dapat meningkatkan kecepatan komputasi. Konsep paralelisasi pada OpenCL pada dasarnya adalah semua *thread* menjalakan instruksi yang sama, namun bagian data yang diproses oleh masing-masing *thread* berbeda-beda. Misalnya operasi penjumlahan dua vektor sepanjang *N* dapat dikerjakan oleh *N thread*, dimana *thread* ke-*i* hanya menjumlahkan elemen ke-*i* dari dua vektor tersebut. Pada OpenCL, unit komputasi atau *thread* ini disebut *work-item* [10].

OpenCL *kernel* dieksekusi dalam suatu *work-space* yang terdiri dari sekumpulan *work-item* yang membentuk struktur satu hingga tiga dimensi. Pada suatu *work-space*, semua *work-item* mengeksekusi OpenCL *kernel* yang sama. *Work-space* dapat dibagi ke dalam beberapa *work-group*. *Work-group* terdiri dari beberapa *work-item* yang membentuk blok satu hingga tiga dimensi. Secara umum, seluruh *work-item* pada *work-space* bekerja secara independen, namun sinkronisasi dapat dilakukan antar *work-item* yang berada pada *work-group* yang sama.

Setiap *work-item* memiliki *identifier* (ID) yang unik. ID adalah bilangan bulat yang lebih besar dari atau sama dengan nol. Setiap *work-item* memiliki dua jenis ID, yaitu *global* ID dan *local* ID. *Global* ID mengidentifikasi *work-item* dalam suatu *work-space*, sedangkan *local* ID mengidentifikasi *work-item* dalam suatu *work-group*. Setiap *work-group* juga memiliki ID yang unik. Saat suatu proses berjalan pada *device*, ID dari *work-item* dan *work-group* yang menjalankan proses tersebut dapat diambil. ID tersebut dapat dimanfaatkan untuk mengatur paralelisasi dari eksekusi OpenCL *kernel*. Gambar 2.6 adalah contoh *work-space* dua dimensi pada OpenCL.




**Gambar 2.6:** Contoh *work-space* dua dimensi. *Work-space* terbagi menjadi empat *work-group* dua dimensi. Semua *work-group* selalu memiliki ukuran yang sama.

Pada OpenCL terdapat batasan terhadap ukuran *work-group* yang dapat digunakan untuk menjalankan suatu OpenCL *kernel*. Terdapat dua jenis batasan, yaitu batasan dari *device* dan batasan dari OpenCL *kernel*. Setiap *mobile* GPU memiliki batasan banyak maksimal *work-item* yang dapat berada dalam satu *work-group*. Misalnya pada beberapa Adreno GPU, banyak maksimal *work-item* dalam satu *work-group* adalah 1024. Setiap OpenCL *kernel* juga memiliki batasan tersendiri yang ditentukan oleh kompleksitas OpenCL *kernel* tersebut. Jika *kernel* semakin kompleks maka banyak maksimal *work-item* dalam satu *work-group* akan semakin kecil. Batasan yang berasal dari OpenCL *kernel* ini selalu lebih kecil atau sama dengan batasan dari *device*.

## 2.7 Jenis Memori pada OpenCL

Setiap *work-item* yang menjalankan OpenCL *kernel* dapat mengakses beberapa jenis memori [10]. Setiap jenis memori memiliki kelebihan dan kekurangan masing-masing dalam hal latensi dan kapasitas. Berikut adalah empat jenis memori yang secara konsep terdapat pada OpenCL.

1. ***Global Memory***. *Global memory* merupakan memori yang dapat diakses oleh seluruh *work-item* pada suatu *work-space*. Memori ini digunakan untuk menyimpan *buffer object*. Memori ini memiliki latensi paling besar di antara empat jenis memori. Meskipun lambat, memori ini memiliki kapasitas yang paling besar dibandingkan jenis memori lain.

2. ***Constant Memory***. *Constant memory* merupakan memori dengan latensi kecil. *Constant memory* digunakan untuk menyimpan data-data yang bersifat

konstan. Argumen-argumen OpenCL *kernel* yang berupa skalar atau vektor disimpan di *constant memory*. Jika suatu data konstan tidak dapat lagi disimpan di *constant memory* karena sudah penuh, maka data akan disimpan di lokasi lain dan dapat menyebabkan latensinya lebih tinggi.

3. ***Local Memory***. *Local memory* merupakan memori yang dapat diakses oleh semua *work-item* yang berada dalam satu *work-group*. Latensi dari memori ini relatif kecil. *Local memory* sering digunakan untuk melakukan *caching* dalam kasus ketika *work-items* dalam suatu *work-group* perlu mengakses data yang sama berkali-kali.

4. ***Private Memory***. Memori ini bersifat *private* untuk suatu *work-item* dan tidak dapat diakses oleh *work-item* lain. Memori ini digunakan untuk menyimpan *private variable*. Jika *private memory* yang berada di *regsiter* sudah penuh, maka *private variable* akan disimpan di lokasi lain sehingga menyebabkan latensinya lebih tinggi.

Pada OpenCL *kernel*, akses memori sering menjadi *bottleneck*. Meminimalkan *read*/*write* terhadap jenis memori yang lambat dan maksimal *read*/*write* terhadap jenis memori yang cepat dapat membantu meningkatkan performa OpenCL *kernel* secara signifikan. Secara umum *local memory* dan *constant memory* sangat disarankan penggunaannya [10].

## 2.8 Tipe Data Vektor pada OpenCL

Tipe data vektor merupakan sebuah tipe data berupa vektor sepanjang $N$ yang mengandung $N$ skalar di dalamnya. Pada OpenCL, contoh tipe data vektor adalah $floatN$ yang merupakan vektor berisi $N$ buah *floating point* [10]. Nilai $N$ yang mungkin pada OpenCL adalah 2, 4, 8, dan 16. Penggunaan tipe data vektor dapat membantu mengurangi biaya operasi *read*/*write* terhadap memori sehingga meningkatkan performa OpenCL *kernel*. Dengan menggunakan tipe data $float4$ misalnya, suatu vektor yang berisi empat buah *floating point* dibaca hanya melalui satu kali instruksi. Nilai $N$ yang disarankan untuk digunakan pada tipe data vektor adalah 4. Pada beberapa jenis GPU, contohnya Adreno, untuk mengakses vektor yang panjangnya lebih dari 4 diperlukan lebih dari satu instruksi *read*/*write*.

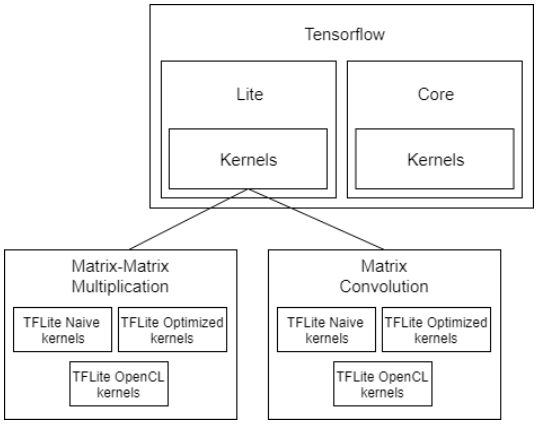
# BAB 3
# METODOLOGI

Pada bagian ini akan dijelaskan metodologi yang digunakan dalam penelitian. Metodologi ini mencakup metode implementasi OpenCL untuk masing-masing operasi serta metode eksperimen.

## 3.1 Metode Implementasi

Pada penelitian ini penulis memanfaatkan Tensorflow Lite untuk menjalankan *Deep Learning inference* pada perangkat *mobile*. Tensorflow Lite memiliki *kernel* tersendiri yang terpisah dari *core* Tensorflow. Penulis memodifikasi Tensorflow Lite *kernel* dengan menambahkan implementasi OpenCL untuk operasi perkalian matriks-matriks, perkalian matriks-vektor, dan konvolusi matriks sehingga operasi-operasi tersebut dapat dijalankan pada GPU ketika *inference* berlangsung. Implementasi OpenCL pada tiga operasi tersebut hanya dapat digunakan oleh *fully-connected* layer dan *convolution layer*.

Tensorflow Lite telah memiliki dua jenis Tensorflow Lite *kernel* untuk operasi matriks pada *convolution layer* dan *fully-connected layer*, yaitu *naive kernel* dan *optimized kernel*, dimana keduanya dijalankan pada CPU. Dengan menambahkan Tensorflow Lite *kernel* yang diimplementasikan melalui OpenCL, ada tiga jenis Tensorflow Lite *kernel* yang dapat digunakan. Saat kompilasi, pengguna dapat memilih untuk menggunakan *naive kernel* (CPU), *optimized kernel* (CPU), atau OpenCL *kernel* (GPU). Gambar 3.1 menunjukkan bagaimana penulis memodifikasi Tensorflow Lite *kernel*.



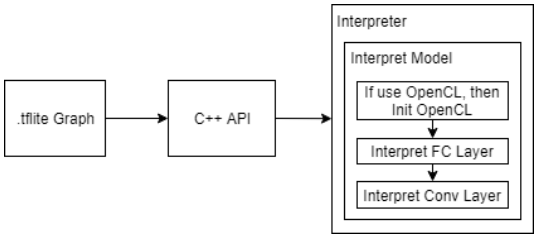

**Gambar 3.1:** Modifikasi Tensorflow Lite *kernel* dengan menambahkan satu jenis *kernel* baru untuk operasi perkalian matriks-matriks dan konvolusi matriks yang diimplementasikan melalui OpenCL dan berjalan di GPU.

Persiapan-persiapan OpenCL seperti yang disebutkan pada BAB II memerlukan cukup banyak waktu. Untuk mengurangi biaya persiapan yang mahal, beberapa persiapan tidak dilakukan pada setiap *inference*, namun hanya dilakukan satu kali pada awal berjalannya aplikasi *Deep Learning*. Hal ini dilakukan dengan cara meletakkan persiapan-persiapan tersebut pada *interpreter* sehingga persiapan tersebut hanya dilakukan ketika Tensorflow Lite melakukan interpretasi model. Objek-objek hasil dari persiapan OpenCL seperti *context* dan *buffer* tersebut kemudian dapat diberikan sebagai argumen ketika melakukan interpretasi *fully-connected layer* dan *convolution layer*. Ketika *inference*, objek *fully-connected layer* dan *convolution layer* pada Tensorflow Lite dapat menggunakan objek-objek OpenCL yang telah dipersiapkan di awal. Persiapan untuk OpenCL yang dilakukan pada awal berjalannya aplikasi dapat dilihat pada Tabel 3.1.

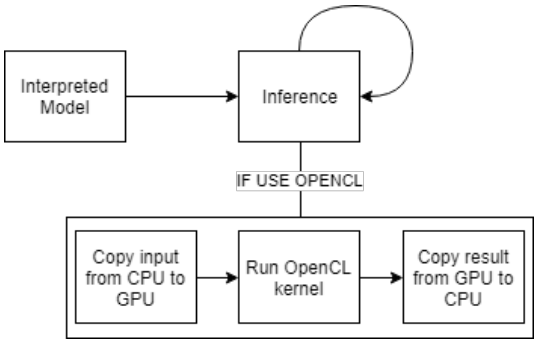
**Tabel 3.1:** Persiapan OpenCL yang dilakukan satu kali pada awal berjalannya aplikasi.

| **No.** | **Persiapan OpenCL** |
|---|---|
| 1 | Membuat *context* |
| 2 | Membuat *command queue* |
| 3 | Membuat *kernel* |
| 4 | Membuat *buffer* |

Persiapan yang tidak dapat dilakukan hanya satu kali adalah menyalin data masukan operasi *inference* dari memori CPU ke memori GPU. Proses menyalin data ini harus dikerjakan pada setiap *inference* karena data masukan bersifat dinamis. Perhatikan Gambar 3.2 dan Gambar 3.3 untuk mengetahui lebih jelas bagaimana persiapan untuk OpenCL dilakukan pada penelitian ini.



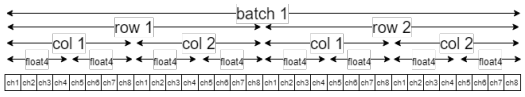

**Gambar 3.2:** Metode persiapan untuk OpenCL yang dilakukan hanya satu kali di awal berjalannya suatu aplikasi *Deep Learning*. Persiapan dilakukan ketika *interpreter* melakukan inisiasi model.




**Gambar 3.3:** Proses menyalin data masukan dan keluaran antara memori CPU dan GPU dilakukan pada setiap *inference*. Proses menyalin data tidak dapat dilakukan satu kali saja karena data bersifat dinamis.

### 3.1.1 Metode Implementasi Konvolusi Matriks

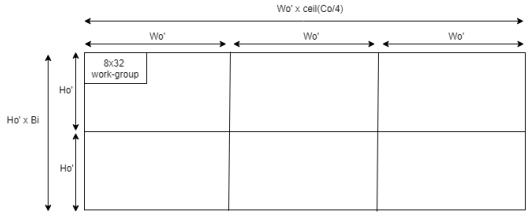
Operasi konvolusi matriks memiliki dua matriks masukan, yaitu *image* dan *filter*, dan satu matriks keluaran yaitu matriks *output*. Matriks-matriks tersebut merupakan matriks empat dimensi yaitu kanal, baris, kolom, dan *batch*. Seluruh matriks masukan dan keluaran disimpan di memori GPU secara linear dengan struktur seperti pada Gambar 3.4. Semua matriks menggunakan tipe data vektor $float4$. Apabila banyaknya kanal dari matriks bukan kelipatan empat, maka diberikan *padding* pada struktur linear di memori GPU tersebut sedemikian sehingga setiap vektor $float4$ mengandung elemen-elemen yang merupakan elemen-elemen matriks pada kolom, baris, dan *batch* yang sama.




**Gambar 3.4:** Struktur linear matriks masukan dan keluaran yang disimpan di memori GPU untuk operasi konvolusi. Elemen ke-15 dari data linear tersebut adalah elemen pada kanal ke-7, kolom ke-2, baris ke-1 dan *batch* ke-1 dari matriks.

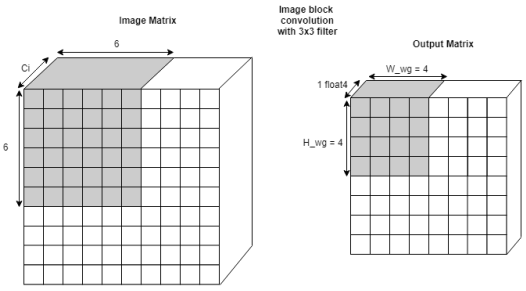
Misalkan banyaknya kanal, tinggi, lebar, dan banyaknya *batch* dari *image*, *filter*, dan *output* berturut-turut adalah $C_i \times H_i \times W_i \times B_i$, $C_f \times H_f \times W_f \times B_f$, dan $C_o \times$

$H_o \times W_o \times B_o$. Pada implementasi ini digunakan *work-space* dua dimensi berukuran $H_{ws} \times W_{ws}$ dan *work-group* dua dimensi berukuran $H_{wg} \times W_{wg}$ dengan $H_{ws} = H'_o \times B_o$ dan $W_{ws} = W'_o \times ceil(C_o/4)$, dimana $Wo'$ adalah bilangan kelipatan $W_{wg}$ terkecil yang lebih besar atau sama dengan $W_o$ dan $H'_o$ adalah bilangan kelipatan $H_{wg}$ terkecil yang lebih besar atau sama dengan $H_o$. Gambar 3.5 merupakan contoh struktur *work-space* dengan $H_{wg} = 8$ dan $W_{wg} = 32$.




**Gambar 3.5:** Struktur *work-space* untuk konvolusi matriks. Dalam kasus ini $W_o$ adalah kelipatan 32 dan $H_o$ adalah kelipatan 8.

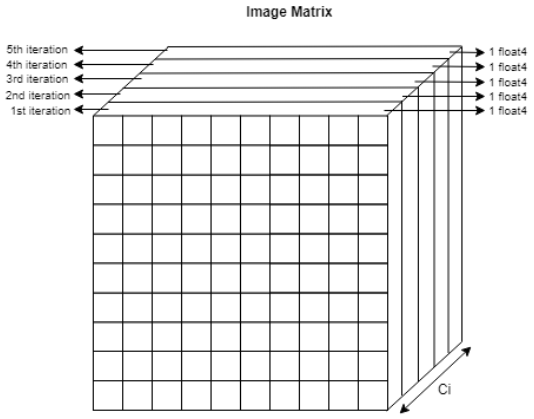
Dengan struktur tersebut, setiap vektor $float4$ pada matriks *output* dikomputasi oleh suatu *work-item* yang unik. Selain itu, suatu *work-group* melakukan komputasi untuk memperoleh satu blok matriks *output* dengan tinggi, lebar, dan banyaknya kanal $H_{wg} \times W_{wg} \times 4$. Blok tersebut merupakan hasil konvolusi dari suatu blok lain pada matriks *image* dengan tinggi, lebar, dan banyaknya kanal $(H_{wg} + H_f - 1) \times (W_{wg} + W_f - 1) \times C_i$ dengan empat *filter* yang berbeda. Ini dapat dilihat pada Gambar 3.6.



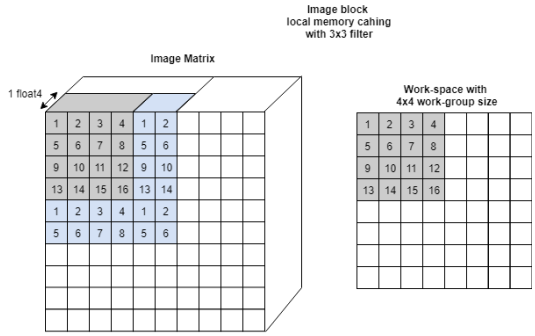

**Gambar 3.6:** Blok pada matriks *output* yang dikomputasi oleh suatu *work-group*. Blok tersebut terdiri dari empat kanal. Blok berwarna abu-abu pada matriks *output* merupakan hasil konvolusi dari blok abu-abu dari matriks *image*.

Perhatikan bahwa untuk memperoleh satu blok matriks *output*, hampir semua elemen pada blok matriks *image* akan diakses lebih dari satu kali oleh *work-group*. Mengetahui fakta ini, penulis menggunakan *local memory caching* untuk mengurangi redundansi akses ke *global memory*. Konvolusi untuk suatu blok matriks *output* dilakukan dalam $ceil(C_i/4)$ iterasi, dimana setiap iterasi hanya melibatkan blok matriks *image* yang berukuran $(H_{wg} + H_f - 1) \times (W_{wg} + W_f - 1) \times 4$ seperti yang

dapat dilihat pada Gambar 3.7. Hasil konvolusi dari semua iterasi kemudian diakumulasikan. Dengan menggunakan *local memory caching*, pada setiap iterasi blok matriks *image* ini disalin terlebih dahulu dari *global memory* ke *local memory* sebelum digunakan untuk komputasi. Setiap *work-item* pada *work-group* bertugas menyalin maksimal empat vektor $float4$ dari blok matriks *image* ke *local memory* seperti yang terlihat pada Gambar 3.8.




**Gambar 3.7:** Operasi konvolusi dilakukan dalam $ceil(C_i/4)$ iterasi dimana $C_i$ adalah kedalaman *image*. Setiap iterasi melibatkan blok matriks *image* dengan kedalaman 4, sesuai dengan panjang vektor $float4$.

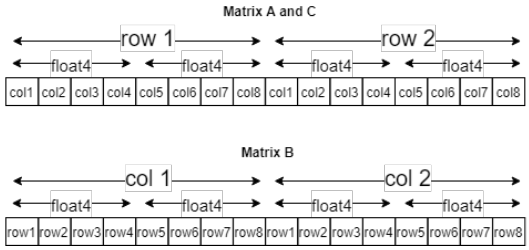



**Gambar 3.8:** *Local memory cahing* terhadap matriks *image* pada suatu iterasi dalam kasus *filter* berukuran panjang dan lebar $3 \times 3$. *Work-item* dengan nomor *i* bertugas menyalin vektor-vektor $float4$ dari *image* dengan nomor *i* ke local memory.

### 3.1.2 Metode Implementasi Perkalian Matriks-Matriks

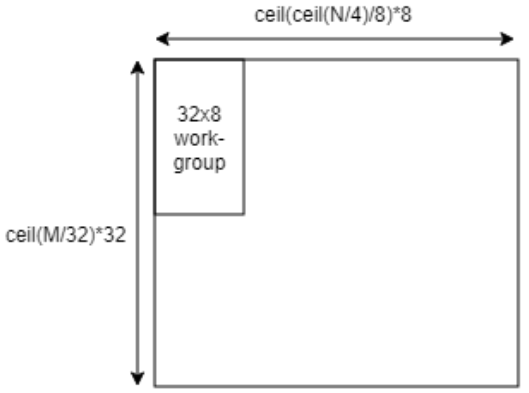
Dalam pembahasan ini dimisalkan matriks masukan adalah matriks A dan matriks B, sedangkan matriks keluaran adalah matriks C. Sama seperti konvolusi, seluruh matriks disimpan secara linear di memori GPU. Matrix A dan matriks C disimpan secara *row-major*, sedangkan matrix B disimpan secara *column-major* seperti pada Gambar 3.9. Semua matriks menggunakan tipe data vektor $float4$. Jika lebar dari

matriks A atau C bukan kelipatan 4, maka diberikan *padding* pada struktur linear di memori GPU sedemikian sehingga setiap vektor $float4$ mengandung elemen-elemen yang merupakan elemen-elemen matriks pada baris yang sama. Untuk matriks B, *padding* diberikan jika tingginya bukan kelipatan 4.




**Gambar 3.9:** Struktur linear matriks A, B, dan C pada operasi perkalian matriks-matriks. A dan C disimpan secara *row-major*, sedangkan B secara *column-major*.

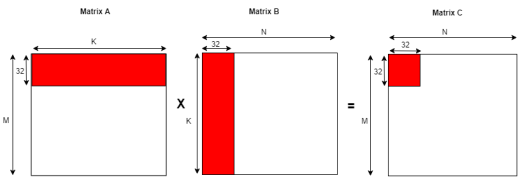
Misalkan matriks A berukuran $M \times K$ dan matriks B berukuran $K \times N$, maka matriks C berukuran $M \times N$. Untuk menjalankan operasi perkalian matriks-matriks di GPU, digunakan *work-space* dua dimensi dengan ukuran $H_{ws} \times W_{ws}$ dan *work-group* dua dimensi berukuran $H_{wg} \times W_{wg}$ dimana $H_{ws}$ adalah bilangan kelipatan $H_{wg}$ terkecil yang lebih besar atau sama dengan $M$ dan $W_{ws}$ adalah bilangan kelipatan $W_{wg}$ terkecil yang lebih besar atau sama dengan $ceil(N/4)$. Untuk ukuran *work-group*, berlaku aturan $H_{wg} = 4 \times W_{wg}$.




**Gambar 3.10:** Struktur *work-space* untuk perkalian matriks-matriks. Dalam kasus ini tinggi *work-space* adalah kelipatan 32 dan lebarnya adalah kelipatan 8.

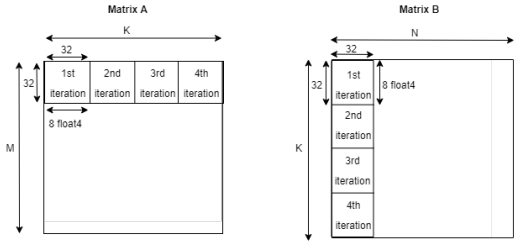
Dengan struktur di atas, setiap vektor $float4$ pada matriks C dikomputasi oleh satu *work-item* yang unik. Lalu, masing-masing *work-group* melakukan komputasi untuk memperoleh satu blok pada matriks C yang berukuran $H_{wg} \times H_{wg}$ seperti pada Gambar 3.10. Perhatikan bahwa setiap blok matriks C tersebut diperoleh dari perkalian matriks antara dua blok lain, yaitu satu blok dari matriks A berukuran $H_{wg} \times K$ dan satu blok dari matriks B berukuran $K \times H_{wg}$. Gambar **??** adalah contoh

perkalian antara dua blok matriks untuk menghasilkan blok berukuran $32 \times 32$ pada matriks C ketika ukuran *work-group* adalah $32 \times 8$.

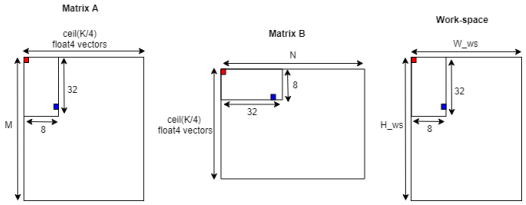



**Gambar 3.11:** Perkalian antara dua blok $32 \times K$ dan $K \times 32$ pada matriks A dan B sehingga menghasilkan satu blok $32 \times 32$ pada matriks C. Ukuran work-group dalam kasus ini adalah $32 \times 8$.

Seperti pada konvolusi, perkalian dua blok matriks tersebut juga mengandung redundansi akses *global memory* karena elemen-elemen yang sama pada blok diakses lebih dari satu kali. Penulis juga menggunakan *local memory caching* dalam implementasi ini. Perkalian antar dua blok matriks A dan B dilakukan dalam $ceil(K/H_{wg})$ iterasi. Pada setiap iterasi, dilakukan perkalian dua blok yang berukuran lebih kecil yaitu antara blok $H_{wg} \times H_{wg}$ dari matriks A dengan blok $H_{wg} \times H_{wg}$ dari matriks B seperti pada Gambar 3.12. Hasil perkalian dari semua iterasi kemudian diakumulasikan. Pada setiap iterasi, masing-masing *work-item* pada *work-group* berutgas menyalin dua vektor $float4$, satu dari blok matriks A dan satu dari blok matriks B, ke *local memory* seperti yang terlihat pada Gambar 3.13 sebelum komputasi dilakukan.




**Gambar 3.12:** Operasi perklaian matriks-matriks yang dilakukan dalam $ceil(K/32)$ iterasi pada kasus ukuran work-group $32 \times 8$. Setiap iterasi melibatkan blok matriks A dengan lebar 32 dan blok matriks B dengan tinggi 32.




**Gambar 3.13:** Pembagian kerja untuk menyalin blok matriks A dan B dari *global memory* ke *local memory* pada *work-group* dengan ukuran $32 \times 8$. Masing-masing *work-item* menyalin dua vektor $float4$. *Work-item* merah memuat vektor berwarna merah dan *work-item* biru memuat vektor berwarna biru.

## 3.2 Metode Eksperimen

Eksperimen dilakukan terhadap masing-masing jenis Tensorflow Lite *kernel* untuk operasi perkalian matriks-matriks dan konvolusi matriks, yaitu Tensorflow Lite *naive kernel* yang berjalan di CPU, Tensorflow Lite *optimized kernel* yang berjalan di CPU, dan Tensorflow Lite OpenCL *kernel* yang berjalan di GPU. Dalam eksperimen ini penulis mengukur kecepatan eksekusi tiga *kernel* tersebut pada berbagai kasus dan ukuran matriks atau vektor masukan. Hasil pengukuran kecepatan dari masing-masing *kernel* kemudian dibandingkan. Untuk Tensorflow Lite OpenCL *kernel*, penulis melakukan dua jenis pengukuran kecepatan. Pertama, pengukuran dilakukan terhadap eksekusi OpenCL *kernel* saja (komputasi saja). Kedua, pengukuran dilakukan terhadap komputasi OpenCL beserta persiapan OpenCL yang diperlukan pada setiap *inference*, yaitu proses menyalin data antar memori CPU dan GPU. Dengan demikian, penulis dapat mengetahui pada apakah *bottleneck* dari OpenCL terletak pada transfer data antar memori atau terletak pada komputasi. Untuk menghitung kecepatan dari suatu Tensorflow Lite *kernel* penulis menggunakan *wall-clock time* dengan cara menghitung selisih dari *wall-clock time* sebelum dan sesudah Tensorflow Lite *kernel* berjalan.

# BAB 4
# EKSPERIMEN DAN ANALISIS

Bagian ini berisi hasil eksperimen terhadap Tensorflow Lite *kernel* baru yang berjalan di GPU yang telah diimplementasikan menggunakan OpenCL. Pada akhir bagian ini diberikan analisis terhadap hasil eksperimen. Terdapat dua operasi yang diuji, yaitu perkalian matriks-matriks dan konvolusi matriks. Eksperimen dilakukan dengan cara memberikan ukuran masukan yang bervariasi terhadap operasi-operasi tersebut. Untuk masing-masing operasi, penulis membandingkan kecepatan dari *OpenCL kernel*, *naive kernel*, dan *optimized kernel*. Kecepatan dari *kernel* diukur menggunakan *wall-clock time*. Penulis menghitung rata-rata kecepatan dari 10 kali eksekusi *kernel* dan juga menghitung standar deviasinya. Satuan yang digunakan untuk mengukur waktu adalah *milisecond*. Eksperimen ini dilakukan pada perangkat Android dengan spesifikasi berikut.

1. CPU : Snapdragon 435, 8x ARM Cortex-A53 @ 1.40Ghz
2. GPU : Adreno 505, OpenCL 2.0 supported
3. RAM : 3GB

## 4.1 Eksperimen Terhadap *Kernel* Operasi Perkalian Matriks-Matriks

Eksperimen ini bertujuan untuk menguji dan membandingkan kecepatan tiga jenis Tensorflow Lite *kernel* untuk operasi perkalian matriks-matriks. Pada eksperimen ini digunakan ukuran yang sama untuk dua matriks masukan dan juga matriks keluaran. Terdapat lima variasi ukuran *panjang* $\times$ *lebar* matriks yang diberikan, yaitu $64 \times 64$, $128 \times 128$, $256 \times 256$, $512 \times 512$, dan $1024 \times 1024$. Dengan demikian, akan terlihat *kernel* mana saja yang unggul dalam komputasi matriks kecil dan *kernel* mana saja yang unggul dalam komputasi matriks besar. Selain itu pada eksperimen ini juga akan terlihat bagaimana persiapan OpenCL berpengaruh terhadap kecepatan Tensorflow Lite *kernel* untuk operasi perkalian matriks-matriks yang diimplementasikan menggunakan OpenCL. Akan diketahui *bottleneck* dari OpenCL pada matriks besar dan matriks kecil.

Hasil eksperimen terhadap kecepatan dari tiga jenis *kernel* dapat dilihat pada Tabel 4.1. Nilai-nilai dari tabel tersebut dalah rata-rata (dalam *milisecond*) dari 10

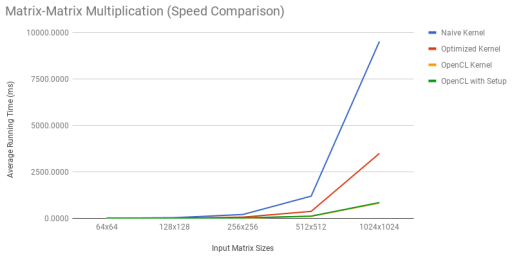
kali eksekusi *kernel*. Standar deviasi dari 10 eksekusi tersebut dapat dilihat pada Tabel 4.2. Secara visual, perbandingan kecepatan antara ketiga jenis *kernel* dapat dilihat pada Gambar 4.1.

**Tabel 4.1:** Hasil eksperimen terhadap Tensorflow Lite *kernel* untuk operasi perkalian matriks-matriks. Waktu dihitung menggunakan satuan detik dengan melakukan rata-rata dari 10 kali *run*. Penyalinan data tidak dihitung untuk menentukan kecepatan.

| Matrix Sizes | Naive Kernel | Optimized Kernel | OpenCL Kernel | OpenCL + Setup |
|---|---|---|---|---|
| 64x64 | 4.3596 | 1.1243 | 1.5501 | 7.6569 |
| 128x128 | 33.9033 | 7.7470 | 2.3841 | 8.6730 |
| 256x256 | 205.9833 | 62.4273 | 14.2164 | 21.7435 |
| 512x512 | 1193.9166 | 373.4296 | 107.3510 | 116.6284 |
| 1024x1024 | 9518.9719 | 3507.8444 | 824.9642 | 848.4501 |

**Tabel 4.2:** Hasil eksperimen terhadap empat kernel operasi perkalian matriks-vektor. Waktu dihitung menggunakan satuan detik dengan melakukan rata-rata dari 10 kali *run*. Penyalinan data tidak dihitung untuk menentukan kecepatan.

| Matrix Sizes | Naive Kernel | Optimized Kernel | OpenCL Kernel | OpenCL + Setup |
|---|---|---|---|---|
| 64x64 | 0.0958 | 0.1270 | 0.0726 | 0.7545 |
| 128x128 | 0.4717 | 0.7324 | 0.0504 | 1.0877 |
| 256x256 | 4.5170 | 1.3869 | 0.0912 | 0.9902 |
| 512x512 | 1.8006 | 3.3557 | 0.1832 | 0.8476 |
| 1024x1024 | 3.1632 | 10.8558 | 0.1265 | 0.9315 |




**Gambar 4.1:** Perbandingan kecepatan empat kernel pada operasi perkalian matriks-vektor tanpa memperhitungkan proses penyalinan data antara CPU dan GPU (ukuran 128x128 hingga 1024x1024).

Dari hasil eksperimen di atas dapat dilihat bahwa Tensorflow Lite *kernel* untuk operasi perkalian matriks-matriks yang berjalan di GPU melalui OpenCL memiliki

performa yang paling baik, terutama untuk matriks berukuran besar. Ketika hanya mempertimbangkan kecepatan komputasi OpenCL *kernel* (tanpa menghitung persiapan), OpenCL *kernel* memiliki performa yang lebih baik dari dua *kernel* lain saat ukuran matriks $128 \times 128$ atau lebih besar. Sementara itu ketika persiapan OpenCL dimasukkan ke dalam penghitungan, OpenCL *kernel* baru mencapai performa terbaik ketika ukuran matriks $256 \times 256$ atau lebih besar.

## 4.2 Eksperimen Terhadap *Kernel* Operasi Konvolusi Matriks

Eksperimen ini bertujuan untuk menguji dan membandingkan kecepatan tiga jenis Tensorflow Lite *kernel* untuk operasi konvolusi matriks. Pada eksperimen ini digunakan beberapa jenis ukuran matriks-matriks masukan dan matriks keluaran. Untuk operasi ini penulis membagi eksperimen ke dalam tiga kasus uji, antara lain kasus ketika ukuran $panjang \times lebar$ bervariasi, kasus ketika ukuran $panjang \times lebar$ bervariasi, dan kasus ketika ukuran $panjang \times lebar$ bervariasi. Pada eksperimen ini akan terlihat *kernel* mana saja yang unggul dalam berbagai kasus yang diberikan. Selain itu juga akan terlihat bagaimana persiapan OpenCL berpengaruh terhadap kecepatan Tensorflow Lite *kernel* untuk operasi konvolusi yang menggunakan OpenCL. Akan diketahui *bottleneck* dari OpenCL pada berbagai kasus uji.

### 4.2.1 Eksperimen Konvolusi dengan Panjang dan Lebar *Image* yang Bervariasi

Pada eksperimen ini, diberikan matriks *image* dengan panjang dan lebar yang bervariasi sehingga dapat diketahui bagaimana besar kecilnya ukuran panjang dan lebar *image* mempengaruhi kecepatan dari tiga jenis *kernel*. Spesifikasi ukuran dari matriks image dan filter yang digunakan dapat dilihat pada Tabel **??**. Terdapat lima variasi ukuran $panjang \times lebar$ matriks *image* yang diberikan, yaitu $32 \times 32$, $64 \times 64$, $128 \times 128$, $256 \times 256$, dan $512 \times 512$. Dalam OpenCL, panjang dan lebar dari *image* terkait dengan panjang dan lebar dari *work-space* yang digunakan dalam eksekusi OpenCL *kernel*. Hal ini telah dijelaskan pada BAB III.

**Tabel 4.3:** Spesifikasi ukuran matriks *image* dan *filter* yang diujikan untuk operasi konvolusi pada kasus panjang dan lebar *image* yang bervariasi.

| Matrix | Channel | Height | Width | Batch |
|---|---|---|---|---|
| Image | 4 | varying | varying | 1 |
| Filter | 4 | 5 | 5 | 4 |

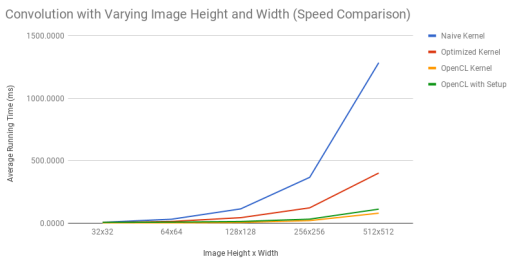
Hasil eksperimen terhadap kecepatan dari tiga jenis *kernel* pada kasus bervariasinya panjang dan lebar *image* ini dapat dilihat pada Tabel 4.3. Nilai-nilai dari tabel tersebut dalah rata-rata (dalam *milisecond*) dari 10 kali eksekusi *kernel*. Standar deviasi dari 10 eksekusi tersebut dapat dilihat pada Tabel 4.4. Secara visual, perbandingan kecepatan antara ketiga jenis *kernel* dapat dilihat pada Gambar 4.2.

**Tabel 4.4:** Hasil eksperimen terhadap empat kernel operasi perkalian matriks-vektor. Waktu dihitung menggunakan satuan detik dengan melakukan rata-rata dari 10 kali *run*. Penyalinan data juga dihitung untuk menentukan kecepatan.

| Image HxW | Naive Kernel | Optimized Kernel | OpenCL Kernel | OpenCL + Setup |
|---|---|---|---|---|
| 32x32 | 6.6511 | 4.7852 | 1.5196 | 6.1364 |
| 64x64 | 31.3152 | 13.4992 | 1.8377 | 7.9563 |
| 128x128 | 113.8935 | 43.4996 | 5.5047 | 13.1798 |
| 256x256 | 366.5030 | 122.3684 | 20.1672 | 31.9092 |
| 512x512 | 1283.9716 | 401.0434 | 79.5439 | 111.8054 |

**Tabel 4.5:** Hasil eksperimen terhadap empat kernel operasi perkalian matriks-vektor. Waktu dihitung menggunakan satuan detik dengan melakukan rata-rata dari 10 kali *run*. Penyalinan data juga dihitung untuk menentukan kecepatan.

| Image HxW | Naive Kernel | Optimized Kernel | OpenCL Kernel | OpenCL + Setup |
|---|---|---|---|---|
| 32x32 | 0.4444 | 0.9186 | 0.1311 | 0.6996 |
| 64x64 | 0.9173 | 2.5479 | 0.0799 | 1.3398 |
| 128x128 | 6.9582 | 9.9099 | 0.2741 | 1.5212 |
| 256x256 | 9.9799 | 4.9044 | 0.3141 | 1.8619 |
| 512x512 | 10.8159 | 3.3544 | 0.1262 | 1.7571 |




**Gambar 4.2:** Perbandingan kecepatan empat kernel pada operasi perkalian matriks-vektor dengan memperhitungkan proses penyalinan data antara CPU dan GPU (ukuran 128x128 hingga 1024x1024).

Dari hasil eksperimen di atas dapat dilihat bahwa Tensorflow Lite *kernel* untuk operasi konvolusi yang berjalan di GPU melalui OpenCL memiliki performa yang paling baik. Ketika hanya mempertimbangkan kecepatan komputasi OpenCL *kernel* (tanpa menghitung persiapan), OpenCL *kernel* memiliki performa yang lebih baik dari dua *kernel* lain pada semua variasi ukuran matriks yang diuji. Sementara itu ketika persiapan OpenCL dimasukkan ke dalam penghitungan, OpenCL *kernel* baru mencapai performa terbaik saat $panjang \times lebar$ dari *image* berukuran $64 \times 64$ atau lebih besar.

### 4.2.2 Eksperimen Konvolusi dengan Banyak Kanal *Image* yang Bervariasi

Pada eksperimen ini, diberikan matriks *image* dengan banyaknya kanal yang bervariasi sehingga dapat diketahui bagaimana banyak sedikitnya kanal dari *image* mempengaruhi kecepatan dari tiga jenis *kernel*. Spesifikasi ukuran dari matriks image dan filter yang digunakan dapat dilihat pada Tabel **??**. Terdapat lima variasi banyaknya kanal matriks *image* yang diberikan, yaitu 64, 128, 256, 512, dan 1024. Dalam OpenCL, banyaknya kanal dari *image* terkait dengan banyaknya iterasi ketika melakukan konvolusi pada setiap *work-item*. Hal ini telah dijelaskan pada BAB III. Semakin besar ukuran kanal dari *image*, semakin besar pula beban komputasi yang dikerjakan oleh suatu *work-item*.

**Tabel 4.6:** Spesifikasi ukuran matriks *image* dan *filter* yang diujikan untuk operasi konvolusi pada kasus banyaknya kanal dari *image* yang bervariasi.

| Matrix | Channel | Height | Width | Batch |
|---|---|---|---|---|
| Image | varying | 16 | 16 | 1 |
| Filter | varying | 5 | 5 | 4 |

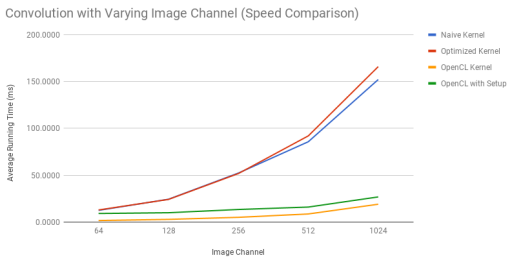
Hasil eksperimen terhadap kecepatan dari tiga jenis *kernel* dalam kasus bervariasinya banyak kanal *image* ini dapat dilihat pada Tabel 4.5. Nilai-nilai dari tabel tersebut dalah rata-rata (dalam *milisecond*) dari 10 kali eksekusi *kernel*. Standar deviasi dari 10 eksekusi tersebut dapat dilihat pada Tabel 4.6. Secara visual, perbandingan kecepatan antara ketiga jenis *kernel* dapat dilihat pada Gambar 4.3.

**Tabel 4.7:** Hasil eksperimen terhadap empat kernel operasi perkalian matriks-vektor. Waktu dihitung menggunakan satuan detik dengan melakukan rata-rata dari 10 kali *run*. Penyalinan data juga dihitung untuk menentukan kecepatan.

| Image Channel | Naive Kernel | Optimized Kernel | OpenCL Kernel | OpenCL + Setup |
|---|---|---|---|---|
| 64 | 12.4833 | 12.8914 | 1.6086 | 9.1080 |
| 128 | 24.3573 | 24.1618 | 2.8356 | 9.9595 |
| 256 | 52.3216 | 51.8085 | 5.0332 | 13.4032 |
| 512 | 85.7740 | 92.0345 | 8.6298 | 15.9954 |
| 1024 | 152.1905 | 165.9758 | 19.0146 | 26.8229 |

**Tabel 4.8:** Hasil eksperimen terhadap empat kernel operasi perkalian matriks-vektor. Waktu dihitung menggunakan satuan detik dengan melakukan rata-rata dari 10 kali *run*. Penyalinan data juga dihitung untuk menentukan kecepatan.

| Image Channel | Naive Kernel | Optimized Kernel | OpenCL Kernel | OpenCL + Setup |
|---|---|---|---|---|
| 64 | 1.2069 | 1.0923 | 0.0874 | 1.1450 |
| 128 | 2.2827 | 1.9532 | 0.3127 | 1.4827 |
| 256 | 1.8923 | 4.9887 | 0.5981 | 1.8302 |
| 512 | 6.2608 | 3.8101 | 0.1258 | 0.8930 |
| 1024 | 2.0152 | 9.3311 | 0.2196 | 0.9503 |




**Gambar 4.3:** Perbandingan kecepatan empat kernel pada operasi perkalian matriks-vektor dengan memperhitungkan proses penyalinan data antara CPU dan GPU (ukuran 128x128 hingga 1024x1024).

Dari hasil eksperimen di atas dapat dilihat bahwa Tensorflow Lite *kernel* untuk operasi konvolusi yang berjalan di GPU melalui OpenCL memiliki performa yang paling baik. Ketika waktu untuk persiapan OpenCL diperhitungkan maupun ketika tidak diperhitungkan, OpenCL *kernel* memiliki performa yang lebih baik dari dua *kernel* lainnya pada semua variasi ukuran matriks yang diuji.

### 4.2.3 Eksperimen Konvolusi dengan Banyak *Batch* dan Kanal *Output* yang Bervariasi

Pada eksperimen ini, diberikan matriks *image* dan *filter* dengan *batch* yang bervariasi. Banyaknya *batch* dari *image* sama dengan banyaknya *batch* dari *output*, sedangkan banyaknya *batch* dari *filter* sama dengan banyaknya kanal dari *output*. Pada eksperimen ini ingin diketahui bagaimana banyak sedikitnya *batch* dan kanal dari *output* mempengaruhi kecepatan tiga jenis kernel *kernel*. Spesifikasi ukuran dari matriks image dan filter yang digunakan dapat dilihat pada Tabel **??**. Terdapat lima variasi ukuran $panjang \times lebar$ matriks *image* yang diberikan, yaitu $8 \times 8$, $16 \times 16$, $32 \times 32$, $64 \times 64$, dan $128 \times 128$. Dalam OpenCL, banyaknya *batch* dan kanal dari matriks *output* terkait dengan lebar dan panjang dari *work-space* yang digunakan dalam eksekusi OpenCL *kernel*. Hal ini telah dijelaskan pada BAB III.

**Tabel 4.9:** Spesifikasi ukuran matriks *image* dan *filter* yang diujikan untuk operasi konvolusi pada kasus banyaknya *batch* dan *kanal* dari *output* yang bervariasi.

| Matrix | Channel | Height | Width | Batch |
|---|---|---|---|---|
| Image | 4 | 16 | 16 | varying |
| Filter | 4 | 5 | 5 | varying |

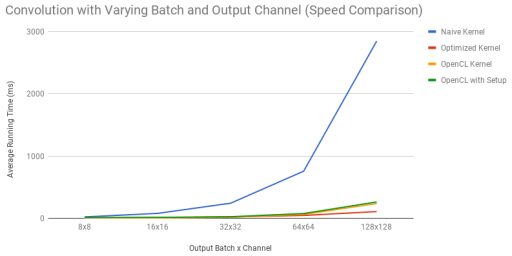
Hasil eksperimen terhadap kecepatan dari tiga jenis *kernel* pada kasus bervariasinya banyak *batch* dan kanal *output* ini dapat dilihat pada Tabel 4.7. Nilai-nilai dari tabel tersebut dalah rata-rata (dalam *milisecond*) dari 10 kali eksekusi *kernel*. Standar deviasi dari 10 eksekusi tersebut dapat dilihat pada Tabel 4.8. Secara visual, perbandingan kecepatan antara ketiga jenis *kernel* dapat dilihat pada Gambar 4.4.

**Tabel 4.10:** Hasil eksperimen terhadap empat kernel operasi perkalian matriks-vektor. Waktu dihitung menggunakan satuan detik dengan melakukan rata-rata dari 10 kali *run*. Penyalinan data juga dihitung untuk menentukan kecepatan.

| Output BxC | Naive Kernel | Optimized Kernel | OpenCL Kernel | OpenCL + Setup |
|---|---|---|---|---|
| 8x8 | 21.3206 | 6.8149 | 1.7034 | 9.6239 |
| 16x16 | 77.6343 | 11.5020 | 4.5229 | 12.9872 |
| 32x32 | 241.5599 | 20.3202 | 15.9239 | 24.1819 |
| 64x64 | 756.2591 | 44.4415 | 61.0587 | 75.0377 |
| 128x128 | 2847.1164 | 106.8449 | 236.1338 | 262.0281 |

**Tabel 4.11:** Hasil eksperimen terhadap empat kernel operasi perkalian matriks-vektor. Waktu dihitung menggunakan satuan detik dengan melakukan rata-rata dari 10 kali *run*. Penyalinan data juga dihitung untuk menentukan kecepatan.

| Output BxC | Naive Kernel | Optimized Kernel | OpenCL Kernel | OpenCL + Setup |
|---|---|---|---|---|
| 8x8 | 2.4153 | 0.6333 | 0.1556 | 1.5570 |
| 16x16 | 4.3455 | 1.6288 | 0.2239 | 1.4405 |
| 32x32 | 7.2629 | 1.5248 | 0.1507 | 1.0567 |
| 64x64 | 3.0043 | 1.2937 | 0.0650 | 2.5150 |
| 128x128 | 8.1991 | 7.3458 | 0.6611 | 10.9892 |




**Gambar 4.4:** Perbandingan kecepatan empat kernel pada operasi perkalian matriks-vektor dengan memperhitungkan proses penyalinan data antara CPU dan GPU (ukuran 128x128 hingga 1024x1024).

Dari hasil eksperimen di atas dapat dilihat bahwa Tensorflow Lite *optimized kernel* yang berjalan di CPU memiliki performa yang paling baik ketika ukuran matriks *output* semakin besar. *Optimized kernel* mampu mengungguli performa OpenCL *kernel* tanpa persiapan ketika $batch \times kanal$ dari *output* berukuran $64 \times 64$ atau lebih besar. *Optimized kernel* juga mampu mengungguli performa OpenCL *kernel* pada semua variasi ukuran matriks *output* yang diuji ketika persiapan OpenCL diperhitungkan.

## 4.3 Analisis

# BAB 5
# KESIMPULAN DAN SARAN

Bagian ini berisi kesimpulan dari penelitian dan saran untuk penelitian-penelitian selanjutnya.

## 5.1 Kesimpulan

## 5.2 Saran

# DAFTAR REFERENSI


[1] Andrew G. Howard and (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision. CoRR, abs/1704.04861.

[2] Christian Szegedy and (2015). Rethinking the Inception Architecture for Computer Vision. CoRR, abs/1512.00567.

[3] Introduction to TensorFlow Lite — TensorFlow. (n.d.). Retrieved from https://www.tensorflow.org/mobile/tflite/.

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

[5] 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

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

[7] LeCun, Y., Haffner, P., Bottou, L., & Bengio, Y. (1999). Object Recognition with Gradient-Based Learning. https://doi.org/10.1007/3-540-46805-6_19.

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

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

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

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

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

[13] 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

# LAMPIRAN

# LAMPIRAN 1