UNIVERSITAS INDONESIA

DEEP LEARNING INFERENCE PADA MOBILE GPU MENGGUNAKAN OPENCL

TUGAS AKHIR

TSESAR RIZQI PRADANA
1406543725

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI ILMU KOMPUTER
DEPOK
JANUARI 2018

UNIVERSITAS INDONESIA

DEEP LEARNING INFERENCE PADA MOBILE GPU MENGGUNAKAN OPENCL

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 Menggunakan OpenCL
Nama	:	Tsesar Rizqi Pradana
NPM	:	1406543725

Laporan Tugas Akhir ini telah diperiksa dan disetujui.

20 Januari 2018

Prof. T. Basaruddin	Risman Adnan
Pembimbing I Tugas Akhir	Pembimbing II 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 Menggunakan OpenCL

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 I	:	Prof. T. Basaruddin	(	)
Pembimbing II	:	Risman Adnan	(	)
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 Menggunakan OpenCL” 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 Menggunakan OpenCL

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 Menggunakan OpenCL

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 ketika *inference*. Penulis menggunakan OpenCL, API pemrograman paralel untuk berbagai jenis prosesor, untuk mengimplementasikan *kernel* tersebut. Hasil pengujian yang dilakukan menunjukkan bahwa OpenCL *kernels* yang berjalan di GPU untuk operasi perkalian matriks-matriks memiliki performa yang lebih baik daripada *naive kernel* dan *optimized kernel* dari Tensorflow Lite yang berjalan di CPU, terutama ketika matriks berukuran besar. Sementara itu, OpenCL *kernel* untuk operasi konvolusi matriks mampu mengungguli performa *naive kernel* dan *optimized kernel* ketika banyaknya *batch* dan kanal dari matriks *output* cukup sedikit.

Kata Kunci:
Deep Learning; Mobile GPU; OpenCL;

ABSTRACT

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

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 ketika *inference*. Penulis menggunakan OpenCL, API pemrograman paralel untuk berbagai jenis prosesor, untuk mengimplementasikan *kernel* tersebut. Hasil pengujian yang dilakukan menunjukkan bahwa OpenCL *kernels* yang berjalan di GPU untuk operasi perkalian matriks-matriks memiliki performa yang lebih baik daripada *naive kernel* dan *optimized kernel* dari Tensorflow Lite yang berjalan di CPU, terutama ketika matriks berukuran besar. Sementara itu, OpenCL *kernel* untuk operasi konvolusi matriks mampu mengungguli performa *naive kernel* dan *optimized kernel* ketika banyaknya *batch* dan kanal dari matriks *output* cukup sedikit.

Keywords:
Deep Learning; Mobile GPU; OpenCL;

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB 1
PENDAHULUAN

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

1.1 Latar Belakang

Deep Learning merupakan teknik *Machine Learning* yang mampu mempelajari representasi data secara otomatis melalui suatu model *Neural Network* yang terdiri dari beberapa lapisan pemrosesan [6]. Model *Deep Learning* dapat menerima masukan berupa data mentah untuk melakukan klasifikasi. Model *Deep Learning* ditingkatkan akurasinya melalui proses latihan dan digunakan untuk memprediksi label atau kelas dari suatu data melalui proses *inference*. Salah satu jenis model *Deep Learning*, *Convolutional Neural Network* (CNN) [18], memiliki peran besar dalam meningkatnya popularitas *Deep Learning* dalam beberapa tahun terakhir. CNN memiliki kemampuan yang sangat baik dalam memproses data berupa citra, video, maupun audio. Selain CNN, dalam *Deep Learning* juga terdapat jenis model lain seperti *Long Short-Term Memory* (LSTM) [13] yang dapat digunakan untuk melakukan pemrosesan data bersifat sekuensial.

Seiring meningkatnya popularitas *Deep Learning*, usaha-usaha untuk meningkatkan kecepatan proses latihan maupun *inference* pada *Deep Learning* terus bermunculan. Saat ini CPU tidak lagi menjadi pilihan utama untuk menjalankan proses latihan maupun *inference* pada komputer personal atau *server*. Perangkat lunak Deep Learning untuk komputer personal atau *server* saat ini sudah dapat menjalankan proses-proses *Deep Learning* di GPU. Proses latihan dan *inference* pada *Deep Learning* mengandung operasi-operasi matriks yang berat seperti perkalian matriks-matriks dan konvolusi matriks. GPU yang memiliki performa komputasi yang tinggi dapat menjalankan operasi-operasi tersebut dengan lebih cepat. Usaha juga dilakukan untuk membawa *Deep Learning* kepada perangkat yang lebih kecil, yaitu perangkat *mobile*. Belum lama ini, Google merilis perangkat lunak sumber terbuka bernama Tensorflow Lite [4] yang memungkinkan pengguna menjalankan proses *inference* pada *Deep Learning* pada perangkat *mobile*.

Tensorflow Lite saat ini hanya dapat menjalankan proses *inference* di CPU.

Penulis melihat peluang penggunaan GPU untuk menjalankan proses *inference* pada Tensorflow Lite, yaitu dengan cara mengimplementasikan program untuk operasi-operasi matriks pada proses *inference* yang berjalan di GPU dan mengintegrasikannya ke Tensorflow Lite. Penulis ingin mengetahui apakah penggunaan *mobile* GPU untuk menjalankan operasi-operasi matriks tersebut dapat meningkatkan kecepatan proses *inference* pada Tensorflow Lite. Menjalankan proses *inference* di *mobile* GPU sebenarnya bukanlah hal yang baru. Sudah ada perangkat lunak bernama CNNdroid [5] yang dapat digunakan untuk menjalankan proses *inference* di GPU untuk model CNN. CNNdroid memanfaatkan Renderscript untuk menjalankan komputasi pada GPU.

Dalam penelitian ini penulis mencoba menggunakan OpenCL untuk mengimplementasikan program untuk operasi-operasi matriks pada proses *inference* yang berjalan di GPU. OpenCL merupakan API pemrograman paralel untuk berbagai jenis prosesor seperti CPU, GPU, dan FPGA [10]. Program yang diimplementasikan menggunakan OpenCL memiliki portabilitas tinggi karena dapat dijalankan pada berbagai prosesor yang berbeda-beda. OpenCL merupakan API pemrograman paralel yang mendukung komputasi dengan menggunakan banyak *thread*. GPU merupakan prosesor dengan ratusan *core* yang mendukung komputasi secara paralel. Penggunaan *mobile* GPU untuk menjalankan operasi-operasi matriks pada proses *inference* diharapkan dapat meningkatkan kecepatan proses *inference* pada Tensorflow Lite.

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 matriks pada proses *inference* di GPU pada Tensorflow Lite?

2. Bagaimana perbandingan kecepatan operasi-operasi matriks pada proses *inference* antara yang berjalan di GPU (melalui OpenCL) dengan yang berjalan di CPU (implementasi asli Tensorflow Lite)?

1.2.2 Batasan Permasalahan

Berikut adalah batasan dan asumsi pada penelitian.

1. Penulis hanya mengimplementasikan dua operasi matriks pada proses *inference* menggunakan OpenCL, yaitu operasi perkalian matriks-matriks dan operasi konvolusi matriks.

2. Program OpenCL yang telah diimplementasikan hanya dapat digunakan untuk perangkat *Android* saja.

3. Operasi konvolusi matriks yang diimplementasikan menggunakan OpenCL diasumsikan memiliki jangkah sebesar satu.

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 operasi-operasi matriks pada proses *inference* pada Tensorflow Lite menggunakan OpenCL sehingga dapat berjalan di GPU.

2. Membandingkan kecepatan operasi-operasi matriks pada proses *inference* antara yang berjalan di GPU (melalui OpenCL) dengan yang berjalan di CPU (implementasi asli Tensorflow Lite)?

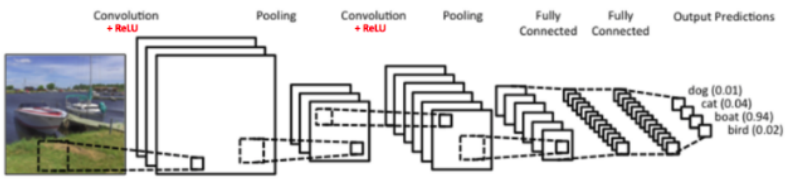
BAB 2
LANDASAN TEORI

Bagian ini berisi penjelasan mengenai teori-teori dan pengetahuan yang terkait dengan pelaksanaan penelitian.

2.1 Deep Learning

Deep Learning merupakan teknik *Machine Learning* yang menggunakan model *Neural Network* yang terdiri dari beberapa lapisan pemrosesan untuk melakukan klasifikasi. Teknik *Machine Learning* konvensional memerlukan representasi data (fitur-fitur data) sebagai masukan untuk melakukan klasifikasi. Pada *Deep Learning*, model dapat menerima masukan berupa data mentah. Model dapat secara otomatis mempelajari dan menghasilkan representasi data yang diperlukan untuk melakukan klasifikasi. Oleh karena itu *Deep Learning* merupakan salah satu bentuk dari *Representation Learning*. Pada suatu model *Deep Learning* terdapat lapisan-lapisan yang bertugas untuk mempelajari dan menghasilkan representasi data dan terdapat lapisan-lapisan yang bertugas untuk melakukan klasifikasi [6].

Contoh model *Deep Learning* yang populer adalah *Convolutional Neural Network* (CNN). CNN dapat memproses data berupa citra, video, maupun audio. CNN tersusun dari beberapa lapisan pemrosesan yang masing-masing terdiri dari beberapa *node*. Pada bagian awal model terdapat lapisan konvolusi yang berfungsi untuk menghasilkan fitur-fitur dari data dengan cara melakukan konvolusi menggunakan suatu filter. Lapisan konvolusi dapat disambungkan dengan lapisan *pooling* yang berfungsi untuk melakukan *downsampling* terhadap data. Pada bagian akhir model biasanya terdapat lapisan *fully-connected*, lapisan yang setiap *node*-nya terhubung dengan setiap *node* pada lapisan tetangganya. Lapisan ini bertugas untuk melakukan klasifikasi menggunakan fitur-fitur data yang diperoleh dari lapisan-lapisan sebelumnya [6]. Gambar 2.1 merupakan contoh model CNN.

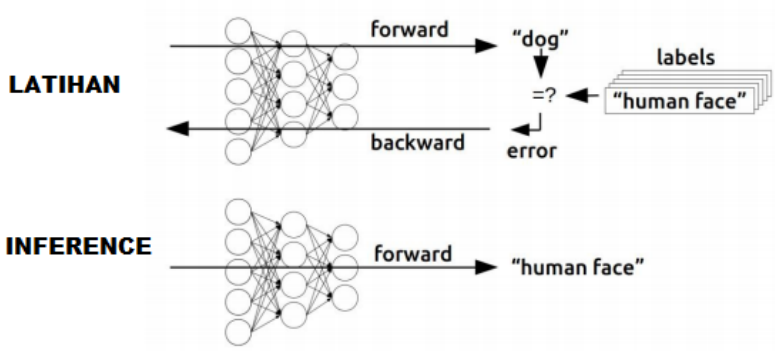


Gambar 2.1: Contoh model *Convolutional Neural Network* yang mengandung lapisan konvolusi, lapisan *pooling*, dan lapisan *fully-connected*.

2.2 Proses Latihan dan *Inference* pada Deep Learning

Lapisan-lapisan pada model *Deep Learning* memiliki suatu parameter yang digunakan untuk menghitung nilai dari masing-masing *node* pada lapisan. Nilai dari parameter tersebut dapat diatur melalui proses latihan. Proses latihan bertujuan untuk memperbarui nilai parameter pada setiap lapisan dengan harapan akurasi model dalam melakukan klasifikasi akan meningkat. Salah satu metode latihan adalah dengan menggunakan sekumpulan data yang telah diketahui labelnya (*Supervised Learning*). Model melakukan prediksi terhadap label dari data-data tersebut dengan melakukan *forward propagation*. Label hasil prediksi kemudian dibandingkan dengan label asli yang telah diketahui. Informasi galat dari hasil prediksi kemudian dikirimkan ke semua lapisan, mulai dari lapisan paling akhir hingga lapisan paling awal (*backward propagation*). Galat tersebut digunakan untuk memperbarui nilai parameter pada setiap lapisan dengan tujuan untuk mengurangi galat pada prediksi-prediksi selanjutnya [6].

Proses *inference* sedikit berbeda dengan proses latihan. Proses *inference* bertujuan untuk memprediksi label dari suatu data (pada umumnya data baru yang belum diketahui labelnya) tanpa melakukan pembaruan nilai parameter. Pada proses ini hanya terjadi *forward-propagation*. Komputasi dilakukan terhadap data mulai dari lapisan pertama, kemudian diteruskan ke lapisan kedua, dan seterusnya hingga mencapai lapisan terakhir. Data pada lapisan terakhir adalah hasil dari *forward-propagation* yang selanjutnya digunakan untuk menentukan label atau kelas dari data [11]. Gambar 2.1 menunjukkan perbedaan proses latihan dan *inference*.



Gambar 2.2: Perbedaan proses latihan dan *inference* pada *Deep Learning*.

2.3 Operasi-operasi Deep Learning Inference

Proses *inference* pada *Deep Learning* melibatkan beberapa operasi matriks yang cukup berat, dua di antaranya adalah operasi konvolusi matriks dan operasi

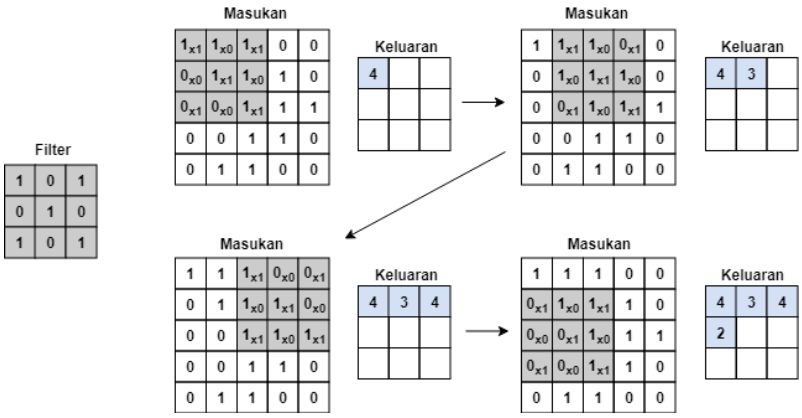
perkalian matriks-matriks. Dua operasi inilah yang pada penelitian ini akan diimplementasikan menggunakan OpenCL dan dijalankan di *mobile* GPU.

2.3.1 Konvolusi Matriks

Operasi konvolusi matriks terjadi pada lapisan konvolusi. Operasi ini melibatkan tiga buah matriks yaitu matriks masukan, matriks filter, dan matriks keluaran. Matriks-matriks tersebut memiliki tiga dimensi yaitu kanal, baris, dan kolom. Banyaknya kanal menyatakan kedalaman matriks, banyaknya baris menyatakan ketinggian matriks, dan banyaknya kolom menyatakan lebar matriks.

Pada lapisan konvolusi, terjadi operasi konvolusi antara matriks filter dan matriks masukan. Suatu matriks masukan dapat dikonvolusikan dengan satu atau lebih filter. Karena itu, filter juga dapat dipandang sebagai matriks yang memiliki empat dimensi yaitu kanal, baris, kolom, dan *batch*. Banyaknya *batch* menyatakan banyaknya filter tiga dimensi (kanal, baris, dan kolom). Konvolusi dari satu *batch* pada filter menghasilkan satu kanal tersendiri pada matriks keluaran, sehingga banyaknya *batch* pada filter akan selalu sama dengan banyaknya kanal (kedalaman) dari matriks keluaran.

Pada operasi konvolusi, posisi filter dimulai dari ujung kiri atas dari matriks masukan. Filter kemudian bergeser ke kanan dan kebawah dengan besar jangkah yang ditentukan (pada umumnya besarnya jangkah adalah satu). Pada suatu posisi filter, dilakukan perkalian titik antara matriks filter dengan submatriks dari matriks masukan yang bersesuaian dengan posisi filter. Karena terjadi perkalian titik, maka kedalaman matriks masukan pasti selalu sama dengan kedalaman matriks filter. Satu kali operasi perkalian titik tersebut menghasilkan skalar yang merupakan satu elemen dari matriks keluaran. Setelah proses konvolusi selesai untuk satu *batch* filter, akan terbentuk satu kanal pada matriks keluaran [9]. Gambar 2.3 menunjukkan contoh operasi konvolusi dengan matriks masukan berukuran 5×5 (satu kanal), filter berukuran 3×3 (satu kanal dan satu *batch*), dan jangkah sebesar satu.

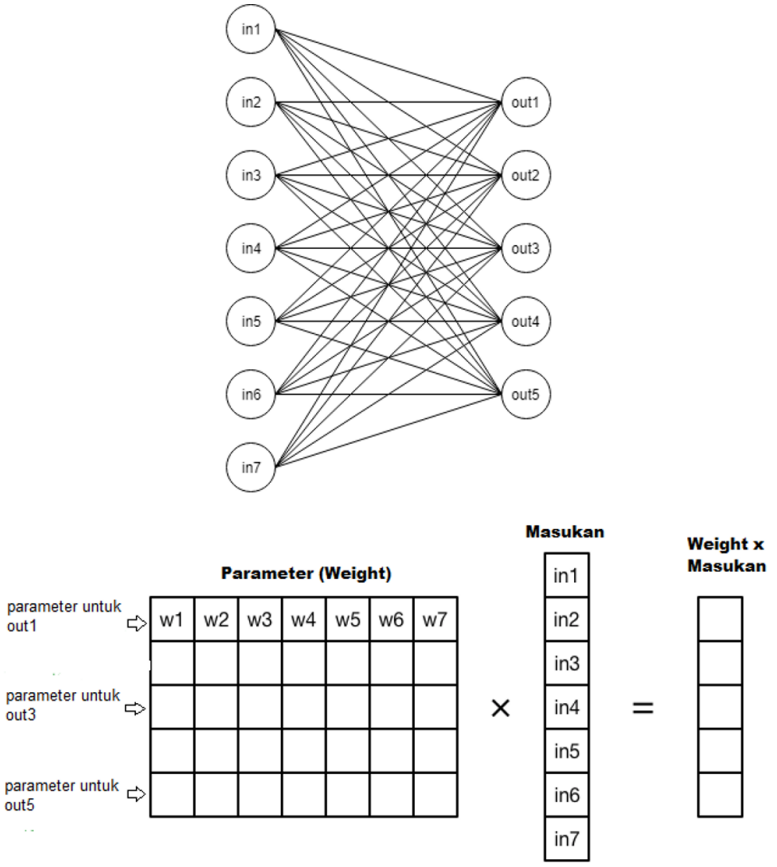


Gambar 2.3: Contoh operasi konvolusi.

Suatu operasi konvolusi dapat dilakukan sekaligus terhadap lebih dari satu matriks masukan dan menghasilkan lebih dari satu matriks keluaran. Jika demikian, maka matriks masukan dan matriks keluaran juga dapat dianggap sebagai matriks yang memiliki empat dimensi seperti matriks filter dengan dimensi tambahan yaitu *batch*.

2.3.2 Perkalian Matriks-Matriks

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



Gambar 2.4: Ilustrasi perkalian matriks-vektor pada lapisan *fully-connected*, dimana elemen ke-i pada vektor $weight \times input$ adalah nilai out_i.

Operasi perkalian matriks berukuran $M \times K$ dengan vektor berukuran $K \times 1$ pada lapisan *fully-connected* tersebut dapat dilakukan sekaligus untuk N *batch*, sehingga akan terjadi operasi perkalian matriks-matriks antara matriks berukuran $M \times K$ dengan matriks berukuran $K \times N$ yang menghasilkan matriks berukuran $M \times N$.

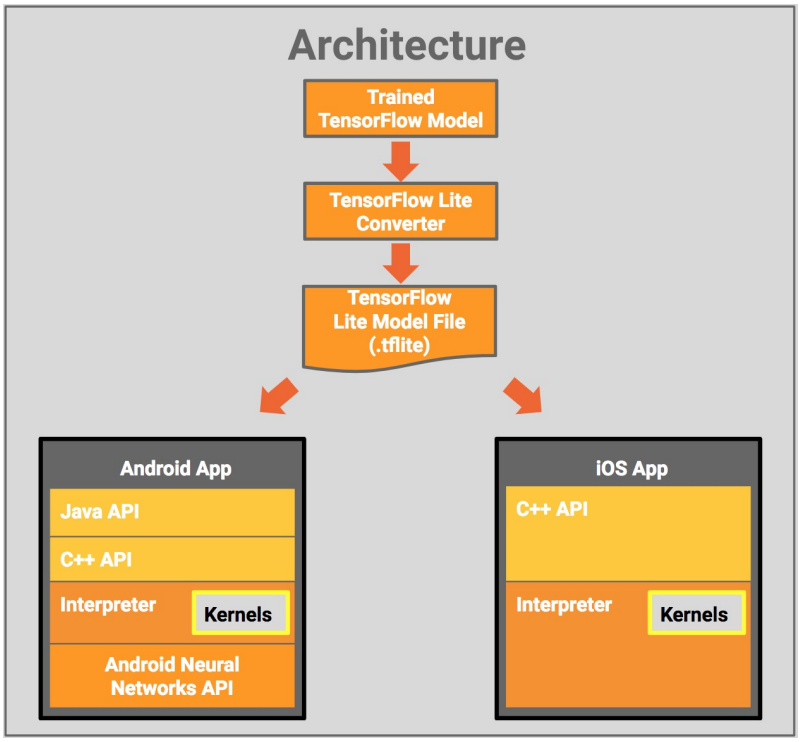
Selain lapisan *fully-connected*, operasi perkalian matriks-matriks juga dapat terjadi pada lapisan konvolusi, yaitu ketika operasi konvolusi melibatkan filter yang berukuran $tinggi \times lebar = 1 \times 1$. Matriks masukan dari operasi konvolusi dapat dipandang sebagai matriks dua dimensi dengan 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 matriks masukan. 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 matriks filter.

2.4 Tensorflow Lite

TensorFlow [8] adalah perangkat lunak sumber terbuka yang merupakan pustaka untuk melakukan komputasi numerik menggunakan graf. *Node* pada graf merepresentasikan operasi matematika, sedangkan *edge* pada graf merepresentasikan *tensor* (larik multidimensi) sebagai masukan atau keluaran dari operasi pada *node*. Tensorflow dikembangkan oleh para pengembang dan peneliti dari Google untuk keperluan penelitian di bidang *Machine Learning* and *Deep Learning*. Selain dapat digunakan pada komputer personal atau *server*, Tensorflow juga digunakan pada perangkat *mobile* untuk menjalankan proses *inference* melalui pustaka Tensorflow Lite.

Tensorflow Lite [4] merupakan pustaka *Machine Learning* terbaru dari Tensorflow yang ditujukan untuk perangkat Android dan IoS. Tensorflow Lite diimplementasikan menggunakan bahasa C/C++. Saat ini Tensorflow Lite memiliki dukungan untuk menjalankan proses *inference* pada model CNN dan LSTM. Tensorflow Lite memiliki *kernel* tersendiri yang terpisah dari pusat Tensorflow. Yang dimaksud dengan *kernel* pada Tensorflow Lite adalah program yang digunakan untuk menjalankan proses *inference* pada *Deep Learning*. Contoh *kernel* pada Tensorflow Lite adalah *kernel* perkalian matriks-matriks dan *kernel* konvolusi matriks. Tensorflow Lite memiliki suatu jenis *kernel* yang telah dioptimalkan untuk perangkat *mobile*, yaitu *optimized kernel*, yang memiliki performa yang sangat baik. Selain *optimized kernel* terdapat pula jenis *kernel* biasa, yaitu *naive kernel*, yang performanya tidak lebih baik dari *optimized kernel*.

Untuk melakukan proses *inference*, Tensorflow Lite memerlukan graf dengan format ".tflite" yang berisi model *Deep Learning*. Graf tersebut akan diinterpretasi oleh *interpreter* pada Tensorflow Lite. *Interpreter* akan memuat semua *kernel* yang diperlukan oleh model pada graf tersebut secara selektif. Gambar 2.5 menunjukkan arsitektur Tensorfow Lite.



Gambar 2.5: Arsitektur Tensorflow Lite dengan *interpreter* yang bertugas memuat *kernel* yang diperlukan secara selektif.

2.5 OpenCL

OpenCL merupakan API untuk melakukan pemrograman paralel pada prosesor yang berbeda-beda seperti CPU, GPU, dan FPGA. OpenCL dapat digunakan untuk meningkatkan performa komputasi secara signifikan. OpenCL API tersedia dalam bahasa C/C++ dengan ekstensi. Pada OpenCL terdapat dua sisi program, yaitu *host* dan *device*. *Device* adalah prosesor target tempat berjalannya komputasi. *Host* adalah yang mengatur jalannya komputasi pada *device*. Menyalin data masukan dari memori *host* ke memori *device* dan menentukan banyaknya *thread* yang bekerja adalah contoh tugas dari *host*. OpenCL *device* pada penelitian ini adalah GPU. Dalam suatu program OpenCL terdapat istilah-istilah yang perlu dipahami sebagai berikut [1].

1. ***Kernel***. *Kernel* pada OpenCL adalah serangkaian instruksi yang mendefinisikan suatu fungsi tertentu, sama seperti fungsi pada C/C++. *Kernel* dieksekusi pada *device*, namun dikompilasi dan dijadwalkan eksekusinya oleh *host*.

2. ***Buffer***. *Buffer* merupakan objek memori yang menyimpan koleksi data secara linear dalam *bytes*. *Buffer* berada pada memori *device* (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*.

3. ***Command Queue***. *Command queue* merupakan objek yang menampung perintah-perintah yang akan dieksekusi pada *device*. OpenCL *kernel* dijadwalkan eksekusinya melalui *command queue*.

4. ***Context***. *Context* adalah lingkungan dimana komputasi pada *device* dilakukan. Pada *context* didefiniskan *kernel* yang digunakan, *device* yang digunakan, *buffer* yang dapat diakses oleh *kernel*, dan *command queue* yang digunakan.

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*. Data-data yang terkait dengan eksekusi OpenCL *kernel* (misalnya matriks masukan dan matriks keluaran) perlu disalin secara manual dari memori *host* ke memori *device* 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*.

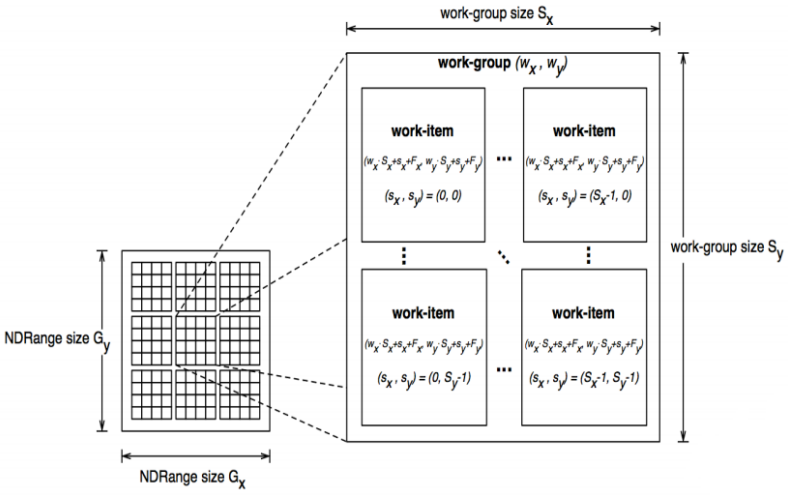
Untuk mengimplementasikan OpenCL, diperlukan pustaka OpenCL (”libOpenCL.so”) dan OpenCL *headers* (”cl.h” dan ”cl_platform.h”). Pada NVIDIA GPU, pustaka OpenCL dapat diperoleh dalam paket instalasi CUDA. Pada perangkat Android, pustaka OpenCL disediakan oleh vendor dari masing-masing prosesor. Perangkat Android dengan vendor GPU Adreno atau Mali biasanya sudah dilengkapi dengan pustaka OpenCL yang terletak pada direktori ”/system/ven-

dor/lib/”. Pustaka tersebut dapat diambil menggunakan perintah ”adb pull” pada Linux.

2.6 Paralelisasi pada OpenCL

Kernel pada OpenCL dapat dijalakan oleh banyak *thread*. Semua *thread* menjalakan *kernel* yang sama, namun dapat dibuat sedemikian sehingga bagian data yang diproses oleh setiap *thread* berbeda. Ini adalah teknik pemrograman paralel yang umum digunakan pada OpenCL. Misalnya operasi penjumlahan dua buah vektor sepanjang *N* dapat dikerjakan oleh *N thread*, dimana *thread* ke-*i* hanya menjumlahkan elemen ke-*i* dari dua vektor tersebut. Pada OpenCL, *thread* ini disebut *work-item*. Seluruh *work-item* yang mengeksekusi suatu *kernel* membentuk blok dalam suatu ruang *N* dimensi yang disebut *NDRange*. Nilai *N* yang valid hingga saat ini adalah 1, 2, dan 3. Semua *work-item* pada suatu *NDRange* dikelompokkan ke dalam *work-group*. *Work-group* terdiri dari beberapa *work-item* yang membentuk blok satu hingga tiga dimensi [1].

Setiap *work-item* memiliki pengidentifikasi (ID) yang unik. ID adalah bilangan bulat yang lebih besar dari atau sama dengan nol. Setiap *work-item* memiliki dua jenis ID, yaitu ID global dan ID lokal. ID global mengidentifikasi *work-item* dalam suatu *NDRange*, sedangkan ID lokal 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 dan dapat dimanfaatkan untuk mengatur paralelisasi dari eksekusi OpenCL *kernel* [10]. Gambar 2.6 adalah contoh *NDRange* dua dimensi pada OpenCL.



Gambar 2.6: Contoh *NDRange* dua dimensi yang terdiri dari beberapa *work-group* dua dimensi.

Pada GPU, semua *work-item* dalam suatu *work-group* bekerja secara konkuren pada suatu unit komputasi. Sinkronisasi dapat dilakukan antara semua *work-item* yang berada pada suatu *work-group* yang sama. Hal ini merupakan inti dari paralelisasi pada OpenCL. OpenCL hanya menjamin bahwa *work-item* yang berada dalam suatu *work-group* yang sama bekerja secara konkuren. Tidak dapat diasumsikan bahwa semua *work-item* dalam suatu *NDRange* selalu bekerja secara konkuren, meskipun hal tersebut sebenarnya sering terjadi. Dengan demikian, penggunaan *work-group* yang berukuran besar sangat disarankan karena dapat meningkatkan kecepatan eksekusi *kernel* [1].

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*. 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* [10].

2.7 Jenis Memori pada OpenCL

Setiap *work-item* yang menjalankan OpenCL *kernel* dapat mengakses beberapa jenis memori. 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 [10].

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

2. **Memori Konstan**. Memori konstan merupakan memori dengan latensi kecil namun kapasitasnya tidak sebesar memori global. Memori konstan digunakan untuk menyimpan data-data yang bersifat konstan. Argumen-argumen OpenCL *kernel* yang berupa skalar atau vektor adalah contoh data yang disimpan di memori konstan.

3. **Memori Lokal**. Memori lokal merupakan memori yang dapat diakses oleh

semua *work-item* yang berada dalam satu *work-group*. Latensi dari memori ini relatif kecil. Memori lokal sering digunakan untuk melakukan *caching* dalam kasus ketika beberapa *work-item* dalam suatu *work-group* perlu mengakses data yang sama berkali-kali. Kapasitas memori ini juga tidak sebesar memori global.

4. **Memori Pribadi**. Memori ini bersifat pribadi untuk suatu *work-item* dan tidak dapat diakses oleh *work-item* lain. Memori ini digunakan untuk menyimpan variabel-variabel pribadi yang dibuat dalam sebuah *kernel*.

Optimisasi memori merupakan salah satu teknik paling penting dan paling efektif dalam meningkatkan performa *kernel*. Pada OpenCL *kernel*, akses memori sering menjadi *bottleneck* [15]. Meminimalkan operasi baca/tulis terhadap jenis memori yang lambat dan maksimal operasi baca/tulis terhadap jenis memori yang cepat dapat membantu meningkatkan performa OpenCL *kernel* secara signifikan. Secara umum memori lokal dan memori konstan sangat disarankan penggunaannya.

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 spesifikasi OpenCL 1.2, 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 baca/tulis 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 baca/tulis [15].

BAB 3
METODOLOGI

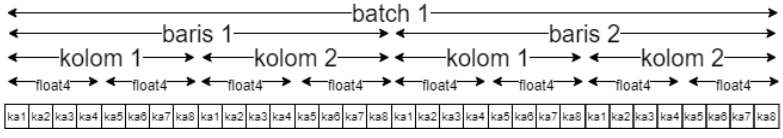
Pada bagian ini akan dijelaskan metodologi yang digunakan dalam penelitian. Metodologi ini mencakup metode implementasi dan metode eksperimen.

3.1 Metode Implementasi

Bagian ini menjelaskan bagaimana penulis mengimplementasikan operasi-operasi matriks pada proses *inference* menggunakan OpenCL dan bagaimana penulis mengintegrasikan hasil implementasi tersebut ke Tensorflow Lite sehingga dapat digunakan oleh Tensorflow Lite ketika melakukan proses *inference*.

3.1.1 Metode Implementasi Konvolusi Matriks

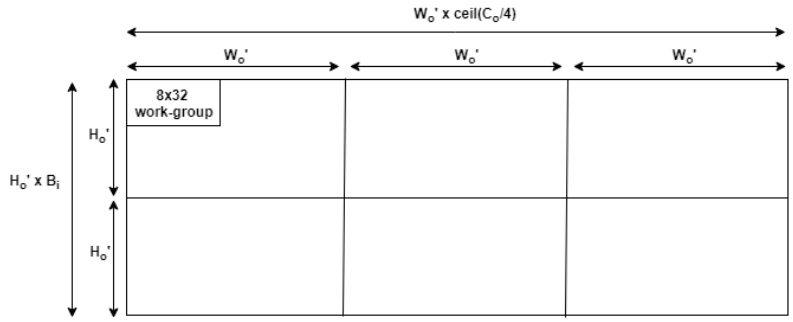
Operasi konvolusi matriks melibatkan tiga buah matriks yaitu matriks masukan, matriks filter, dan matriks keluaran. Matriks-matriks tersebut merupakan matriks empat dimensi yaitu kanal, baris, kolom, dan *batch*. Seluruh matriks masukan dan keluaran disimpan di memori GPU secara linear dengan struktur seperti contoh pada Gambar 3.1. Pada contoh tersebut, elemen ke-15 dari data linear adalah elemen pada kanal ke-7, kolom ke-2, baris ke-1 dan *batch* ke-1 dari matriks. 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 3.1: Contoh struktur linear matriks masukan, matriks filter, dan keluaran yang disimpan di memori GPU untuk operasi konvolusi.

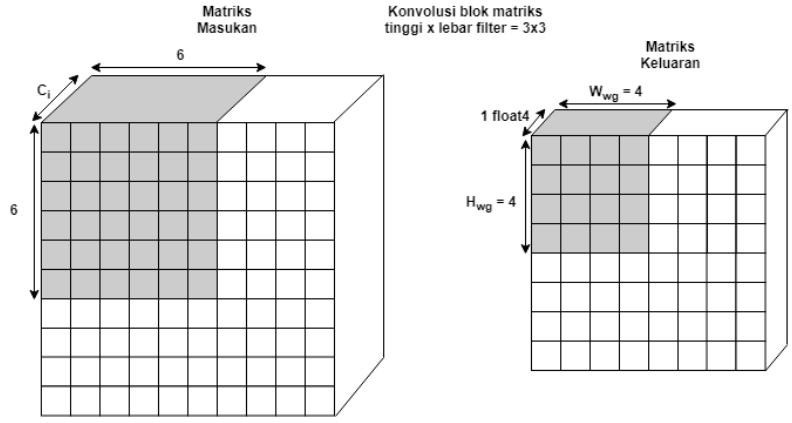
Misalkan ukuran $tinggi \times lebar \times kanal \times batch$ dari matriks masukan, matriks filter, dan matriks keluaran berturut-turut adalah $H_i \times W_i \times C_i \times B_i$, $H_f \times W_f \times C_f \times B_f$, dan $H_o \times W_o \times C_o \times B_o$. Pada implementasi ini digunakan *NDRange* 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 3.2 merupakan contoh struktur *NDRange* dengan $H_{wg} = 8$ dan $W_{wg} = 32$.



Gambar 3.2: Contoh struktur *NDRange* untuk operasi konvolusi matriks.

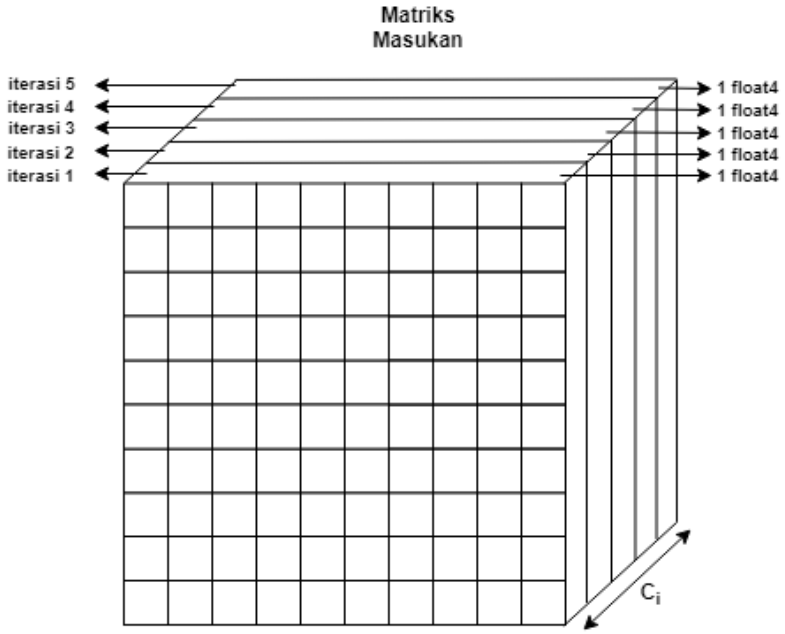
Dengan struktur tersebut, setiap vektor $float4$ pada matriks keluaran dikomputasi oleh suatu *work-item* yang unik. Selain itu, suatu *work-group* melakukan komputasi untuk memperoleh satu blok matriks keluaran dengan ukuran $kanal \times tinggi \times lebar = 4 \times H_{wg} \times W_{wg}$. Blok tersebut merupakan hasil konvolusi dari suatu blok lain pada matriks masukan dengan ukuran $kanal \times tinggi \times lebar = C_i \times (H_{wg} + H_f - 1) \times (W_{wg} + W_f - 1)$ dengan empat matriks filter yang berbeda. Ini dapat dilihat pada Gambar 3.3.



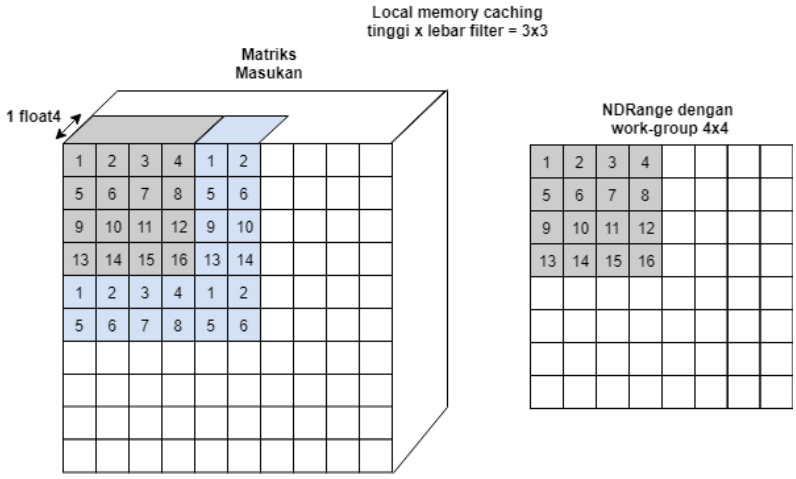
Gambar 3.3: Blok pada matriks keluaran (berwarna abu-abu) yang merupakan hasil konvolusi dari blok (berwarna abu-abu) pada matriks masukan yang dikomputasi oleh suatu *work-group*.

Perhatikan bahwa untuk memperoleh satu blok matriks keluaran, hampir semua elemen pada blok matriks masukan akan diakses lebih dari satu kali oleh *work-group*. Mengetahui fakta ini, penulis menggunakan *local memory caching* untuk mengurangi redundansi akses ke memori global. Konvolusi untuk suatu

blok matriks keluaran dilakukan dalam $ceil(C_i/4)$ iterasi, dimana setiap iterasi hanya melibatkan blok matriks masukan yang berukuran $kanal \times tinggi \times lebar = 4 \times (H_{wg} + H_f - 1) \times (W_{wg} + W_f - 1)$ seperti yang dapat dilihat pada Gambar 3.4. Hasil konvolusi dari semua iterasi kemudian diakumulasikan. Dengan menggunakan *local memory caching*, pada setiap iterasi blok matriks masukan ini disalin terlebih dahulu dari memori global ke memori lokal sebelum digunakan untuk komputasi. Setiap *work-item* pada *work-group* bertugas menyalin maksimal empat vektor $float4$ dari blok matriks masukan ke memori lokal seperti yang terlihat pada Gambar 3.5.

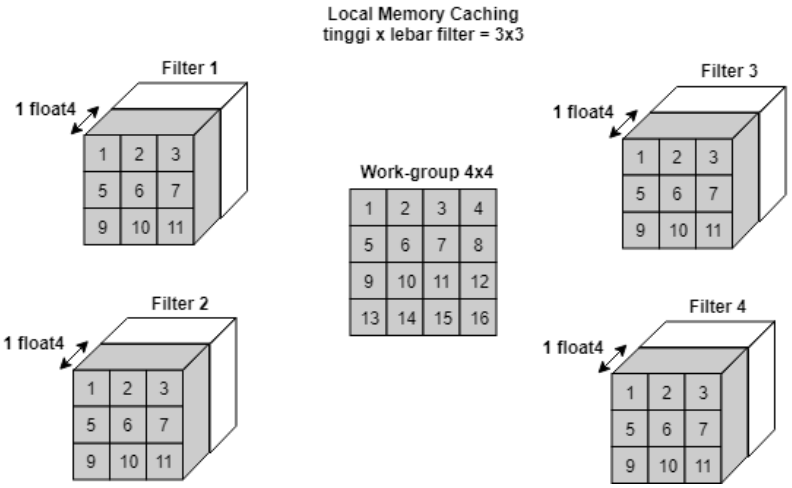


Gambar 3.4: Operasi konvolusi yang dilakukan dalam $ceil(C_i/4)$ iterasi dimana C_i adalah kedalaman matriks masukan.



Gambar 3.5: Contoh *local memory caching* terhadap matriks masukan pada suatu iterasi dimana *work-item* dengan nomor i bertugas menyalin vektor-vektor $float4$ dari matriks masukan dengan nomor i ke memori lokal.

Selain melakukan *local memory caching* terhadap blok pada matriks masukan, *local memory caching* juga dilakukan terhadap blok matriks filter. Blok matriks filter yang dimaksud adalah blok pada matriks filter yang dikalikan titik dengan blok matriks masukan yang telah disimpan di memori lokal. Blok ini berukuran $kanal \times tinggi \times lebar = 4 \times H_f \times W_f$. Untuk menghasilkan suatu kanal pada matriks keluaran, semua *work-item* pada *work-group* menggunakan blok matriks filter yang sama. Karena suatu *work-group* memproses blok matriks keluaran dengan kanal sebanyak empat, maka pada setiap iterasi *caching* dilakukan terhadap empat blok matriks filter yang berbeda seperti yang terlihat pada Gambar 3.6. Setiap *work-item* pada *work-group* bertugas menyalin maksimal empat vektor $float4$ dari matriks filter yang berada di memori global ke memori lokal.

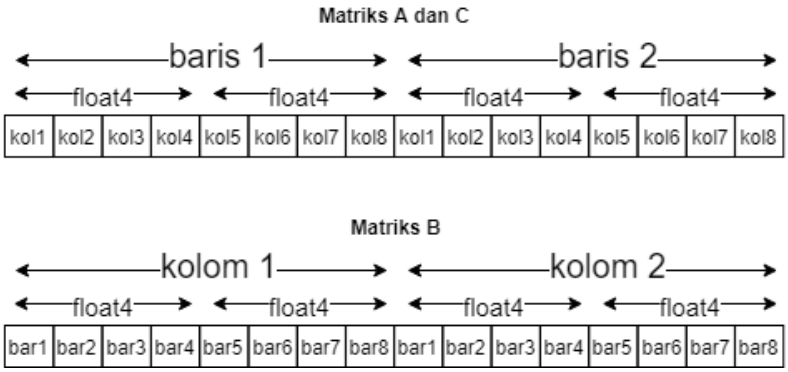


Gambar 3.6: Contoh *local memory caching* terhadap blok matriks filter yang terkait dengan blok matriks masukan pada suatu iterasi, dimana *work-item* dengan nomor *i* bertugas menyalin vektor-vektor $float4$ dari matriks filter dengan nomor *i* ke memori lokal.

Perhatikan bahwa penggunaan metode *local memory caching* seperti yang telah dijelaskan di atas mewajibkan ukuran $tinggi \times lebar$ dari *work-group* harus sama dengan atau lebih besar dari ukuran $tinggi \times lebar$ dari matriks filter.

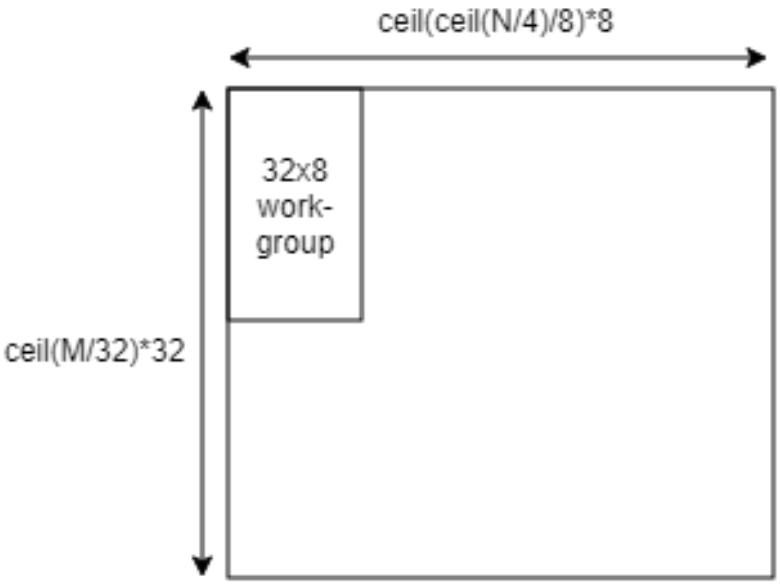
3.1.2 Metode Implementasi Perkalian Matriks-Matriks

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 3.7. 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 3.7: Contoh struktur linear matriks A, B, dan C pada operasi perkalian matriks-matriks.

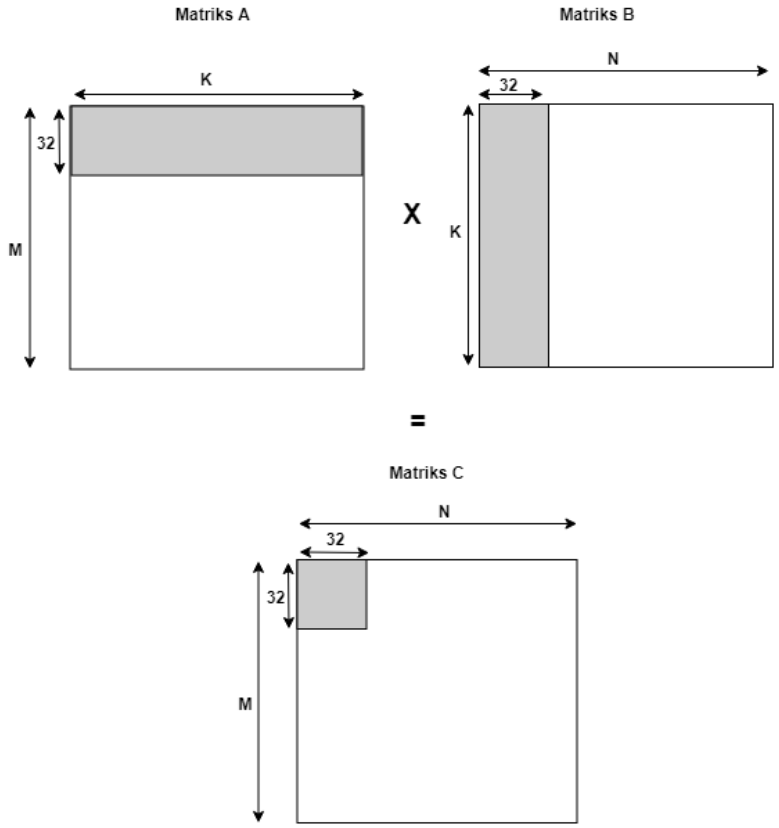
Misalkan ukuran $tinggi \times lebar$ dari matriks A adalah $M \times K$ sedangkan matriks B adalah $K \times N$, maka matriks C berukuran $M \times N$. Untuk menjalankan operasi perkalian matriks-matriks di GPU, digunakan *NDRange* dua dimensi dengan ukuran $H_{ws} \times W_{ws}$ dan *work-group* dua 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 3.8: Contoh struktur *NDRange* untuk perkalian matriks-matriks.

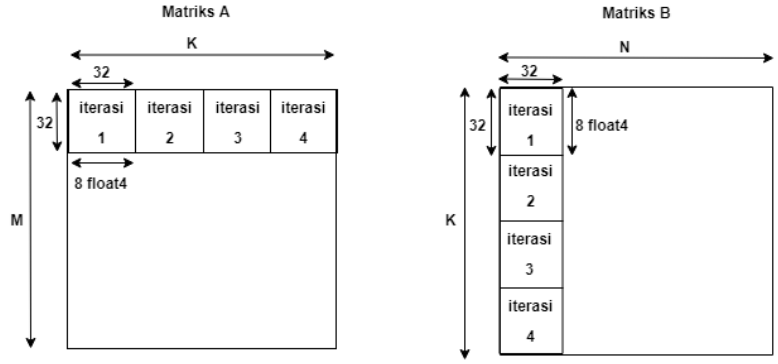
Dengan struktur di atas, setiap vektor $float4$ pada matriks C dikomputasi oleh satu *work-item* yang unik. Lalu, masing-masing *work-group* melakukan komputasi untuk memperoleh satu blok pada matriks C yang berukuran $H_{wg} \times H_{wg}$ seperti pada Gambar 3.8. Perhatikan bahwa setiap blok matriks 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 32×32 pada matriks C ketika ukuran *work-group* adalah 32×8.

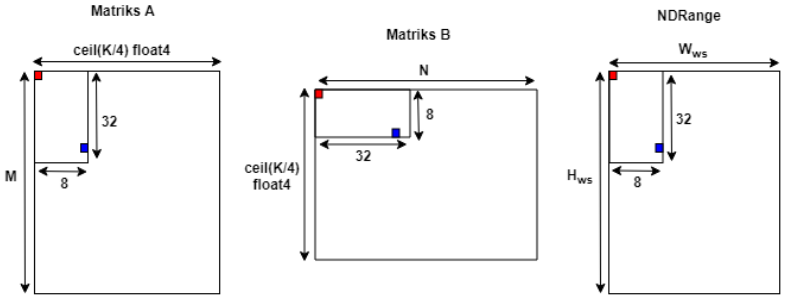


Gambar 3.9: Perkalian antara dua blok berukuran $32 \times K$ dan $K \times 32$ pada matriks A dan matriks B sehingga menghasilkan satu blok berukuran 32×32 pada matriks C.

Seperti pada konvolusi, perkalian dua blok matriks tersebut juga mengandung redundansi akses terhadap memori global karena elemen-elemen yang sama pada blok diakses lebih dari satu kali dalam suatu *work-group*. Penulis juga menggunakan metode *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 3.10. 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 memori lokal seperti yang terlihat pada Gambar 3.11 sebelum komputasi dilakukan.



Gambar 3.10: Operasi perklaian matriks-matriks yang dilakukan dalam $ceil(K/32)$ iterasi dimana setiap iterasi melibatkan blok matriks A dengan lebar 32 dan blok matriks B dengan tinggi 32.



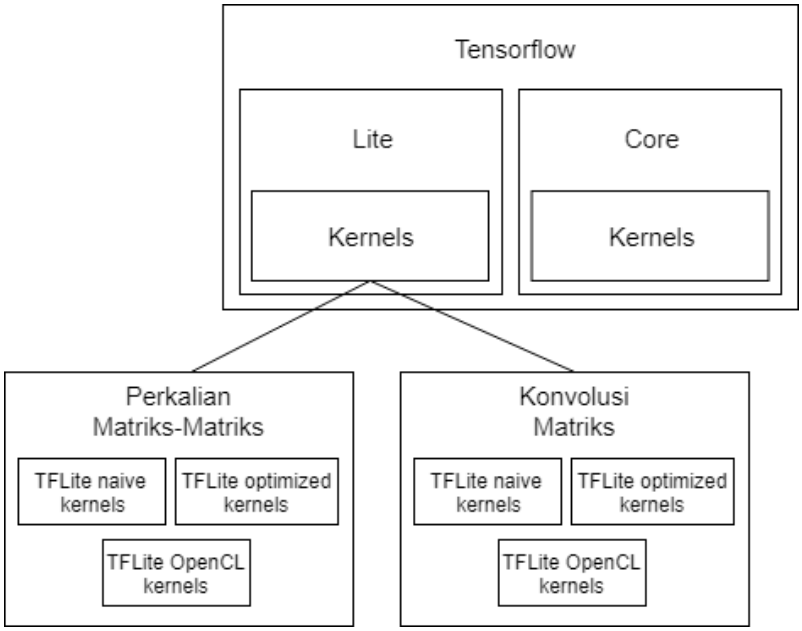
Gambar 3.11: Contoh *local memory caching* pada perkalian matriks-matriks, dimana masing-masing *work-item* menyalin dua vektor $float4$ (*work-item* merah memuat vektor berwarna merah dan *work-item* biru memuat vektor berwarna biru).

3.1.3 Metode Integrasi OpenCL *Kernel* ke Tensorflow Lite

Pada penelitian ini penulis memanfaatkan Tensorflow Lite untuk menjalankan proses *inference* pada *Deep Learning* pada perangkat *mobile*. Penulis memodifikasi kode sumber Tensorflow Lite dengan menambahkan *kernel* untuk operasi perkalian matriks-matriks dan konvolusi matriks yang diimplementasikan menggunakan OpenCL sehingga operasi-operasi tersebut berjalan di GPU ketika proses *inference* berlangsung. *Kernel* tersebut diimplementasikan menggunakan spesifikasi OpenCL versi 1.2, namun juga dapat dijalankan pada perangkat yang memiliki versi OpenCL lebih baru dari 1.2.

Tensorflow Lite telah memiliki dua jenis *kernel* untuk dua operasi matriks tersebut, yaitu *naive kernel* dan *optimized kernel*, dimana keduanya dijalankan pada CPU. Dengan menambahkan *kernel* yang diimplementasikan menggunakan OpenCL, ada tiga jenis Tensorflow Lite *kernel* yang dapat dipilih oleh pengguna untuk digunakan pada proses *inference*. Gambar 3.12 menunjukkan bagaimana penulis memodifikasi kode sumber Tensorflow Lite untuk menambahkan OpenCL

kernel.

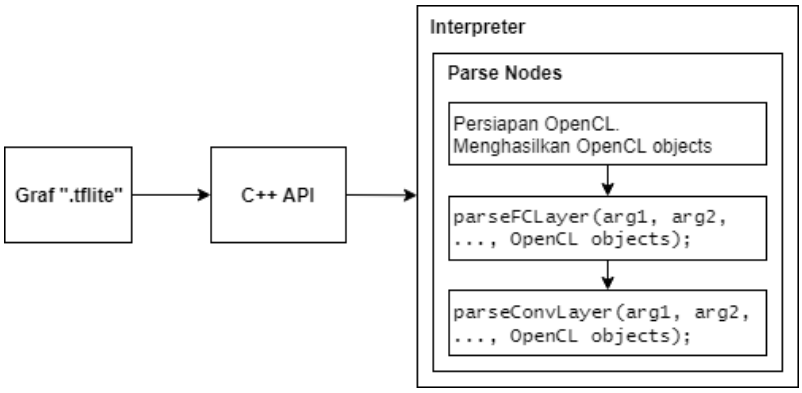


Gambar 3.12: Skema modifikasi kode sumber Tensorflow Lite dengan menambahkan satu jenis *kernel* baru untuk operasi perkalian matriks-matriks dan konvolusi matriks yang diimplementasikan melalui OpenCL dan berjalan di GPU.

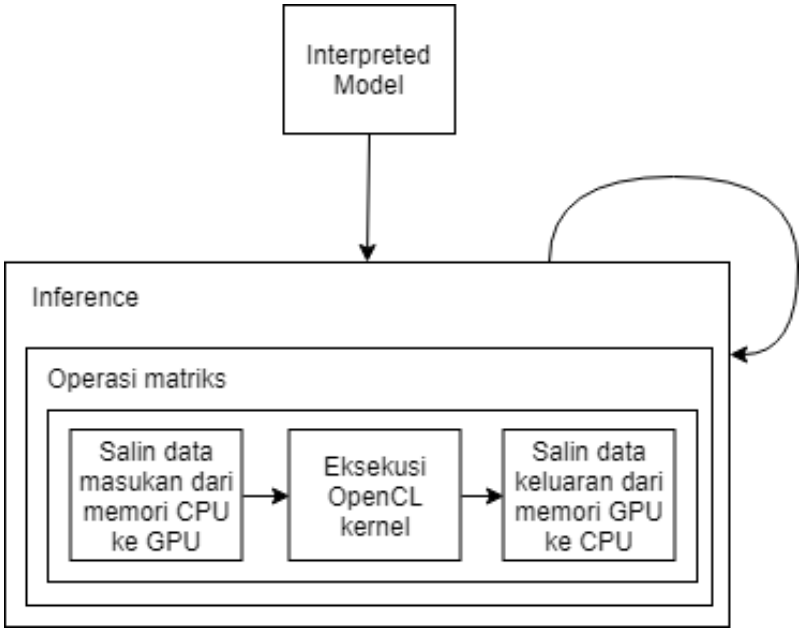
Persiapan-persiapan OpenCL seperti yang disebutkan pada Bagian 2.5 memerlukan cukup banyak waktu. Untuk mengurangi biaya persiapan yang mahal, beberapa persiapan tidak dilakukan pada setiap proses *inference*, namun hanya dilakukan satu kali pada awal berjalannya aplikasi *Deep Learning*. Hal ini dilakukan dengan cara meletakkan persiapan-persiapan tersebut pada *interpreter* Tensorflow Lite. Persiapan tersebut hanya dilakukan ketika *interpreter* akan melakukan *node parsing* terhadap graf Tensorflow Lite. *Node parsing* adalah proses dimana *interpreter* membuat objek-objek *node* berdasarkan graf masukan, termasuk di dalamnya adalah objek lapisan *fully-connected* dan objek lapisan konvolusi. Objek-objek hasil dari persiapan OpenCL seperti *context* dan *buffer* kemudian dapat diberikan sebagai argumen ketika *interpreter* melakukan *parsing* terhadap *node* lapisan *fully-connected* dan lapisan konvolusi. Ketika proses *inference* berlagsung, objek lapisan *fully-connected* dan lapisan konvolusi pada Tensorflow Lite dapat menggunakan objek-objek OpenCL yang telah dipersiapkan di awal.

Persiapan untuk OpenCL yang dilakukan pada awal berjalannya aplikasi adalah membuat *context*, membuat *command queue*, membuat *kernel*, dan membuat *buffer*. Persiapan yang tidak dapat dilakukan hanya satu kali adalah menyalin data masukan operasi matriks dari memori CPU ke memori GPU. Proses menyalin data ini harus selalu dikerjakan pada setiap proses *inference* karena data masukan bersifat

dinamis. Perhatikan Gambar 3.13 dan Gambar 3.14 untuk mengetahui lebih jelas bagaimana persiapan untuk OpenCL dilakukan pada penelitian ini.



Gambar 3.13: Ilustrasi persiapan untuk OpenCL yang dilakukan hanya satu kali di awal berjalannya suatu aplikasi *Deep Learning*.



Gambar 3.14: Ilustrasi proses transfer data antara memori CPU dan GPU yang dilakukan pada setiap proses *inference*.

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 ka-

sus ukuran matriks masukan. Hasil pengukuran kecepatan dari masing-masing *kernel* kemudian dibandingkan. Untuk OpenCL *kernel*, penulis melakukan dua jenis pengukuran kecepatan. Pertama, pengukuran dilakukan terhadap eksekusi OpenCL *kernel* saja, tidak termasuk transfer data antara memori CPU dan GPU. Kedua, pengukuran dilakukan terhadap OpenCL *kernel* beserta transfer data antara memori CPU dan GPU. Dengan demikian, penulis dapat mengetahui apakah *bottleneck* dari program OpenCL yang telah diimplementasikan terletak terletak pada transfer data antar memori atau terletak pada eksekusi OpenCL *kernel*. 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 telah diimplementasikan menggunakan OpenCL. Pada akhir bagian ini diberikan analisis terhadap hasil eksperimen tersebut. 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 setiap *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 milidetik. 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
3. RAM : 3GB

Dengan spesifikasi perangkat tersebut, penulis menggunakan *work-group* berukuran 8×16 untuk menjalankan OpenCL *kernel* pada operasi konvolusi matriks dan *work-group* berukuran 32×8 untuk menjalankan OpenCL *kernel* pada operasi perkalian matriks-matriks. Ukuran *work-group* tersebut merupakan ukuran maksimal yang dapat digunakan pada perangkat tersebut.

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 $tinggi \times lebar$ matriks yang diberikan, yaitu 64×64, 128×128, 256×256, 512×512, dan 1024×1024. 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 transfer data antara memori CPU dan GPU 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.

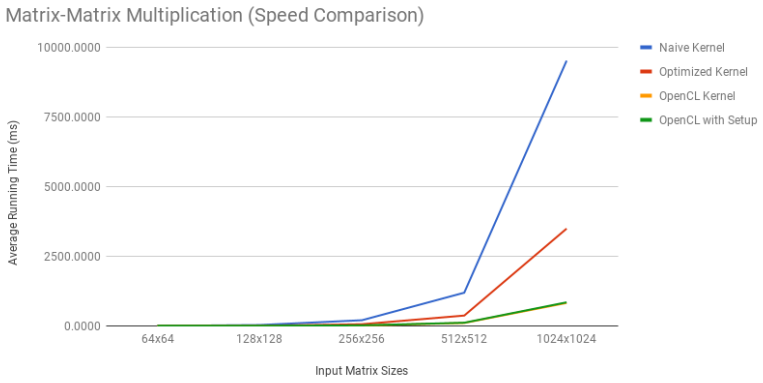
Hasil eksperimen terhadap kecepatan dari tiga jenis *kernel* dapat dilihat pada Tabel 4.1. Nilai-nilai dari tabel tersebut dalah rata-rata (dalam milidetik) dari 10 kali eksekusi *kernel*. Standar deviasi dari 10 eksekusi tersebut dapat dilihat pada Tabel 4.2. Secara visual, perbandingan kecepatan antara ketiga jenis *kernel* dapat dilihat pada Gambar 4.1.

Tabel 4.1: Hasil eksperimen terhadap Tensorflow Lite *kernel* untuk operasi perkalian matriks-matriks, dimana nilai-nilai pada tabel adalah rata-rata dari 10 kali eksekusi dalam milidetik.

Tinggi x Lebar Masukan	**Naive Kernel**	**Optimized Kernel**	**OpenCL Kernel**	**OpenCL + Transfer Data**
64×64	4.3596	1.1243	1.5501	7.6569
128×128	33.9033	7.7470	2.3841	8.6730
256×256	205.9833	62.4273	14.2164	21.7435
512×512	1193.9166	373.4296	107.3510	116.6284
1024×1024	9518.9719	3507.8444	824.9642	848.4501

Tabel 4.2: Standar deviasi dari 10 kali eksekusi (dalam milidetik) Tensorflow Lite *kernel* untuk operasi perkalian matriks-matriks.

Tinggi x Lebar Masukan	**Naive Kernel**	**Optimized Kernel**	**OpenCL Kernel**	**OpenCL + Transfer Data**
64×64	0.0958	0.1270	0.0726	0.7545
128×128	0.4717	0.7324	0.0504	1.0877
256×256	4.5170	1.3869	0.0912	0.9902
512×512	1.8006	3.3557	0.1832	0.8476
1024×1024	3.1632	10.8558	0.1265	0.9315



Gambar 4.1: Perbandingan kecepatan tiga jenis *kernel* untuk operasi perkalian matriks-matriks.

Dari hasil eksperimen di atas dapat dilihat bahwa Tensorflow Lite *kernel* untuk operasi perkalian matriks-matriks yang berjalan di GPU melalui OpenCL memiliki kecepatan yang paling baik, terutama untuk matriks berukuran besar. Ketika hanya memperhatikan kecepatan OpenCL *kernel* tanpa transfer data antar memori, OpenCL *kernel* memiliki kecepatan yang lebih baik dari dua *kernel* lain saat ukuran matriks 128×128 atau lebih besar. Sementara itu ketika transfer data antar memori dimasukkan ke dalam penghitungan, OpenCL *kernel* baru mencapai kecepatan terbaik ketika ukuran matriks 256×256 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 $tinggi \times lebar$ matriks masukan bervariasi, kasus ketika kedalaman matriks masukan bervariasi, dan kasus ketika ukuran $kanal \times batch$ matriks keluaran bervariasi. Pada eksperimen ini akan terlihat *kernel* mana saja yang unggul dalam berbagai kasus yang diberikan. Selain itu juga akan terlihat bagaimana transfer data antara memori CPU dan GPU 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 Tinggi dan Lebar Matriks Masukan yang Bervariasi

Pada eksperimen ini, diberikan matriks masukan dengan tinggi dan lebar yang bervariasi sehingga dapat diketahui bagaimana besar kecilnya ukuran tinggi dan lebar matriks masukan mempengaruhi kecepatan dari tiga jenis *kernel*. Spesifikasi ukuran dari matriks masukan dan matriks filter yang digunakan dapat dilihat pada Tabel 4.3. Terdapat lima variasi ukuran $tinggi \times lebar$ matriks masukan yang diberikan, yaitu 32×32, 64×64, 128×128, 256×256, dan 512×512. Dalam OpenCL, tinggi dan lebar dari matriks masukan terkait dengan tinggi dan lebar dari *NDRange* yang digunakan dalam eksekusi OpenCL *kernel*. Hal ini telah dijelaskan pada BAB III.

Tabel 4.3: Spesifikasi ukuran matriks masukan dan matriks filter yang diujikan untuk operasi konvolusi pada kasus tinggi dan lebar matriks masukan yang bervariasi.

Matriks	**Kedalaman**	**Tinggi**	**Lebar**	**Banyak Batch**
Masukan	4	bervariasi	bervariasi	1
Filter	4	5	5	4

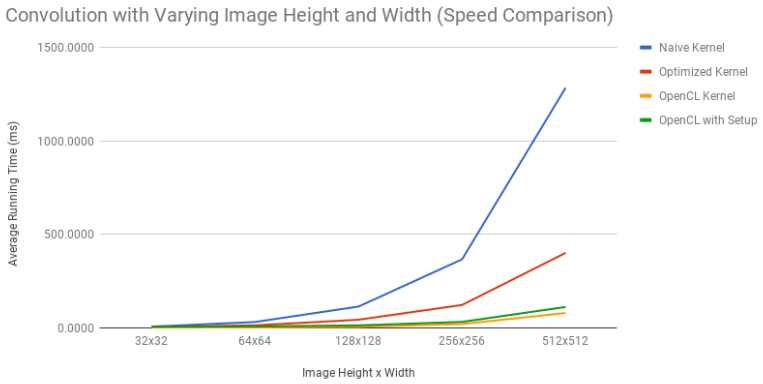
Hasil eksperimen terhadap kecepatan dari tiga jenis *kernel* pada kasus bervariasinya tinggi dan lebar matriks masukan ini dapat dilihat pada Tabel 4.4. Nilai-nilai dari tabel tersebut dalah rata-rata (dalam milidetik) dari 10 kali eksekusi *kernel*. Standar deviasi dari 10 eksekusi tersebut dapat dilihat pada Tabel 4.5. Secara visual, perbandingan kecepatan antara ketiga jenis *kernel* dapat dilihat pada Gambar 4.2.

Tabel 4.4: Hasil eksperimen terhadap Tensorflow Lite *kernel* untuk operasi konvolusi matriks pada kasus ketika tinggi dan lebar matriks masukan bervariasi, dimana nilai-nilai pada tabel adalah rata-rata dari 10 kali eksekusi dalam milidetik.

Tinggi x Lebar Masukan	**Naive Kernel**	**Optimized Kernel**	**OpenCL Kernel**	**OpenCL + Transfer Data**
32×32	6.6511	4.7852	1.5196	6.1364
64×64	31.3152	13.4992	1.8377	7.9563
128×128	113.8935	43.4996	5.5047	13.1798
256×256	366.5030	122.3684	20.1672	31.9092
512×512	1283.9716	401.0434	79.5439	111.8054

Tabel 4.5: Standar deviasi dari 10 kali eksekusi (dalam milidetik) Tensorflow Lite *kernel* untuk operasi konvolusi matriks pada kasus ketika tinggi dan lebar matriks masukan bervariasi.

Tinggi x Lebar Masukan	Naive Kernel	Optimized Kernel	OpenCL Kernel	OpenCL + Transfer Data
32×32	0.4444	0.9186	0.1311	0.6996
64×64	0.9173	2.5479	0.0799	1.3398
128×128	6.9582	9.9099	0.2741	1.5212
256×256	9.9799	4.9044	0.3141	1.8619
512×512	10.8159	3.3544	0.1262	1.7571



Gambar 4.2: Perbandingan kecepatan tiga jenis *kernel* untuk operasi konvolusi matriks pada kasus ketika tinggi dan lebar matriks masukan bervariasi.

Dari hasil eksperimen di atas dapat dilihat bahwa OpenCL *kernel* untuk operasi konvolusi memiliki kecepatan yang paling baik. Ketika transfer data antar memori diabaikan, OpenCL *kernel kernel* memiliki kecepatan yang lebih baik dari dua *kernel* lain pada semua variasi ukuran matriks yang diuji. Sementara itu ketika transfer data antar memori dimasukkan ke dalam penghitungan, OpenCL *kernel* baru mencapai kecepatan terbaik saat $tinggi \times lebar$ dari matriks masukan berukuran 64×64 atau lebih besar.

4.2.2 Eksperimen Konvolusi dengan Kedalaman Matriks Masukan yang Bervariasi

Pada eksperimen ini, diberikan matriks masukan dengan kedalaman yang bervariasi sehingga dapat diketahui bagaimana banyak sedikitnya kanal dari matriks masukan mempengaruhi kecepatan dari tiga jenis *kernel*. Spesifikasi ukuran dari matriks masukan dan matriks filter yang digunakan dapat dilihat pada Tabel 4.6. Terdapat lima

variasi kedalaman matriks masukan yang diberikan, yaitu 64, 128, 256, 512, dan 1024. Dalam OpenCL, kedalaman dari matriks masukan terkait dengan banyaknya iterasi ketika melakukan konvolusi pada setiap *work-item*. Hal ini telah dijelaskan pada BAB III. Semakin besar ukuran kanal dari matriks masukan, semakin besar pula beban komputasi pada setiap *work-item*.

Tabel 4.6: Spesifikasi ukuran matriks masukan dan matriks filter yang diujikan untuk operasi konvolusi pada kasus kedalaman dari matriks masukan yang bervariasi.

Matriks	**Kedalaman**	**Tinggi**	**Lebar**	**Banyak Batch**
Masukan	bervariasi	16	16	1
Filter	bervariasi	5	5	4

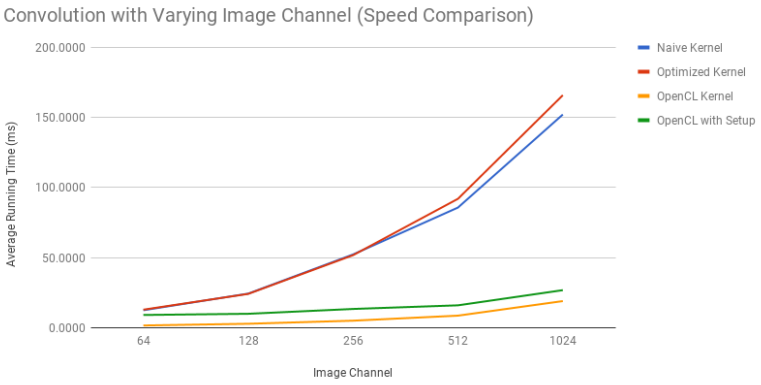
Hasil eksperimen terhadap kecepatan dari tiga jenis *kernel* dalam kasus bervariasinya kedalaman matriks masukan ini dapat dilihat pada Tabel 4.7. Nilai-nilai dari tabel tersebut dalah rata-rata (dalam milidetik) dari 10 kali eksekusi *kernel*. Standar deviasi dari 10 eksekusi tersebut dapat dilihat pada Tabel 4.8. Secara visual, perbandingan kecepatan antara ketiga jenis *kernel* dapat dilihat pada Gambar 4.3.

Tabel 4.7: Hasil eksperimen terhadap Tensorflow Lite *kernel* untuk operasi konvolusi pada kasus ketika kedalaman matriks masukan bervariasi, dimana nilai-nilai pada tabel adalah rata-rata dari 10 kali eksekusi dalam milidetik.

Kedalaman Masukan	**Naive Kernel**	**Optimized Kernel**	**OpenCL Kernel**	**OpenCL + Transfer Data**
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: Standar deviasi dari 10 kali eksekusi (dalam milidetik) Tensorflow Lite *kernel* untuk operasi konvolusi matriks pada kasus ketika kedalaman matriks masukan bervariasi.

Kedalaman Masukan	**Naive Kernel**	**Optimized Kernel**	**OpenCL Kernel**	**OpenCL + Transfer Data**
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 tiga jenis *kernel* untuk operasi konvolusi matriks pada kasus ketika kedalaman matriks masukan bervariasi.

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

4.2.3 Eksperimen Konvolusi dengan Banyaknya *Batch* dan Kedalaman Matriks Masukan yang Bervariasi

Pada eksperimen ini, diberikan matriks masukan dan matriks filter dengan *batch* yang bervariasi. Banyaknya *batch* dari matriks masukan sama dengan banyaknya *batch* dari matriks keluaran, sedangkan banyaknya *batch* dari matriks filter sama dengan kedalaman dari matriks keluaran. Pada eksperimen ini ingin diketahui bagaimana banyak sedikitnya *batch* dan kanal dari matriks keluaran mempengaruhi kecepatan tiga jenis kernel *kernel*. Spesifikasi ukuran dari matriks masukan dan matriks filter yang digunakan dapat dilihat pada Tabel 4.9. Terdapat lima variasi ukuran $tinggi \times lebar$ matriks masukan yang diberikan, yaitu 8×8, 16×16, 32×32, 64×64, dan 128×128. Dalam OpenCL, banyaknya *batch* dan kanal dari matriks keluaran terkait dengan lebar dan tinggi dari *NDRange* yang digunakan dalam eksekusi OpenCL *kernel*. Hal ini telah dijelaskan pada BAB III.

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

Matriks	**Kedalaman**	**Tinggi**	**Lebar**	**Banyak Batch**
Image	4	16	16	bervariasi
Filter	4	5	5	bervariasi

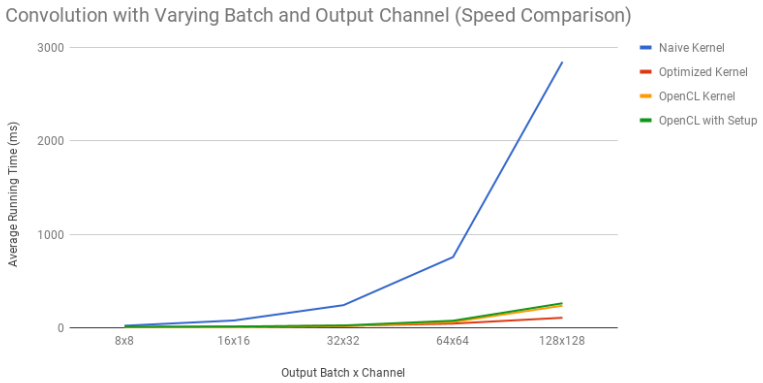
Hasil eksperimen terhadap kecepatan dari tiga jenis *kernel* pada kasus bervariasinya banyak *batch* dan kanal matriks keluaran ini dapat dilihat pada Tabel 4.10. Nilai-nilai dari tabel tersebut dalah rata-rata (dalam milidetik) dari 10 kali eksekusi *kernel*. Standar deviasi dari 10 eksekusi tersebut dapat dilihat pada Tabel 4.11. Secara visual, perbandingan kecepatan antara ketiga jenis *kernel* dapat dilihat pada Gambar 4.4.

Tabel 4.10: Hasil eksperimen terhadap Tensorflow Lite *kernel* untuk operasi konvolusi matriks pada kasus ketika banyaknya *batch* dan kanal matriks keluaran bervariasi, dimana nilai-nilai pada tabel adalah rata-rata dari 10 kali eksekusi dalam milidetik.

Banyak Batch x Kedalaman Masukan	**Naive Kernel**	**Optimized Kernel**	**OpenCL Kernel**	**OpenCL + Transfer Data**
8×8	21.3206	6.8149	1.7034	9.6239
16×16	77.6343	11.5020	4.5229	12.9872
32×32	241.5599	20.3202	15.9239	24.1819
64×64	756.2591	44.4415	61.0587	75.0377
128×128	2847.1164	106.8449	236.1338	262.0281

Tabel 4.11: Standar deviasi dari 10 kali eksekusi (dalam milidetik) Tensorflow Lite *kernel* untuk operasi konvolusi matriks pada kasus ketika banyaknya *batch* dan kanal matriks keluaran bervariasi.

Banyak Batch x Kedalaman Masukan	**Naive Kernel**	**Optimized Kernel**	**OpenCL Kernel**	**OpenCL + Transfer Data**
8×8	2.4153	0.6333	0.1556	1.5570
16×16	4.3455	1.6288	0.2239	1.4405
32×32	7.2629	1.5248	0.1507	1.0567
64×64	3.0043	1.2937	0.0650	2.5150
128×128	8.1991	7.3458	0.6611	10.9892



Gambar 4.4: Perbandingan kecepatan tiga jenis *kernel* untuk operasi konvolusi matriks pada kasus ketika banyaknya *batch* dan kanal matriks keluaran bervariasi.

Dari hasil eksperimen di atas dapat dilihat bahwa Tensorflow Lite *optimized kernel* yang berjalan di CPU memiliki kecepatan yang paling baik ketika ukuran matriks keluaran semakin besar. *Optimized kernel* mampu mengungguli kecepatan OpenCL *kernel* tanpa menghitung transfer data antar memori ketika $batch \times kanal$ dari matriks keluaran berukuran 64×64 atau lebih besar. *Optimized kernel* juga mampu mengungguli kecepatan OpenCL *kernel* pada semua variasi ukuran matriks keluaran yang diuji ketika transfer data antara memori CPU dan GPU diperhitungkan.

4.3 Analisis

Dari eksperimen yang telah dilakukan, dapat dilihat bahwa Tensorflow Lite *kernel* yang berjalan di GPU melalui OpenCL memiliki kecepatan yang cukup baik. OpenCL *kernel* selalu dapat mengungguli kecepatan *naive kernel* dari Tensorflow Lite yang berjalan di CPU pada semua kasus uji baik pada operasi perkalian matriks-matriks maupun konvolusi matriks. OpenCL *kernel* juga mampu mengungguli kecepatan *optimized kernel* pada operasi perkalian matriks-matriks ketika ukuran matriks cukup besar. Pada operasi konvolusi matriks, kecepatan OpenCL *kernel* mampu bersaing dengan *optimized kernel* meskipun terdapat kasus dimana kecepatan OpenCL *kernel* tidak mampu mengungguli kecepatan *optimized kernel*. Hal ini cukup sesuai dengan dugaan awal bahwa operasi-operasi matriks yang diimplementasikan menggunakan paradigma pemrograman paralel dan dijalankan di GPU dapat memiliki kecepatan komputasi yang sangat baik.

Berbeda dengan CPU yang pada umumnya hanya memiliki mepat hingga delapan *cores*, GPU memiliki ratusan *core* yang sangat mendukung komputasi secara paralel sehingga memiliki performa komputasi yang tinggi. Komputasi secara

paralel lebih efisien daripada komputasi sekuensial karena *chips* pada perangkat keras pada dasarnya bersifat paralel. *Chips* mengandung miliaran transistor. Prosesor dengan banyak *core* seperti GPU mengatur transistor-transistor menjadi banyak prosesor paralel yang mengandung ratusan unit komputasi *floating point*. Kemampuan GPU bukan satu-satunya faktor yang meningkatkan performa komputasi, program yang diimplementasikan secara paralel dengan baik juga menjadi faktor dalam meningkatnya kecepatan komputasi. Dalam hal ini OpenCL menyediakan model pemrograman paralel yang baik sehingga mendukung implementasi program paralel yang memiliki performa tinggi [1].

Terdapat dua kasus uji pada eksperimen yang perlu diperhatikan, yaitu ketika tinggi dan lebar matriks masukan bervariasi dan ketika banyaknya *batch* dan kanal matriks keluaran bervariasi. Perhatikan bahwa bertambahnya tinggi dan lebar matriks masukan dan bertambahnya *batch* dan kanal matriks keluaran sama-sama berdampak pada bertambahnya tinggi dan lebar *NDRange*. Artinya, bertambahnya tinggi dan lebar matriks masukan dan bertambahnya *batch* dan kanal matriks keluaran memiliki dampak yang sama terhadap kecepatan OpenCL *kernel*. Dari hasil eksperimen diperoleh bahwa pada saat tinggi dan lebar matriks masukan berukuran besar, kecepatan OpenCL *kernel* mampu mengungguli *optimized kernel*, namun pada saat *batch* dan kanal matriks keluaran berukuran besar, kecepatan *optimized kernel* justru mengungguli OpenCL *kernel*. Hal ini terjadi karena kenaikan waktu eksekusi *optimized kernel* akibat membesarnya *batch* dan kanal matriks keluaran jauh lebih kecil daripada kenaikan waktu eksekusi *optimized kernel* akibat membesarnya tinggi dan lebar matriks masukan.

Penjelasan di atas dapat dibuktikan menggunakan eksperimen tambahan. Pada eksperimen ini penulis membandingkan kecepatan eksekusi pada dua matriks dengan spesifikasi seperti pada Tabel 4.12 dan Tabel 4.13. Pada matriks pertama, tinggi dan lebar matriks masukan berukuran besar. Ukuran $tinggi \times lebar\ NDRange$ yang digunakan oleh OpenCL *kernel* untuk matriks tersebut adalah $(1 \times 512) \times (512 \times 4/4) = 512 \times 512$. Pada matriks kedua, *batch* dan kanal matriks keluaran cukup berukuran besar. Ukuran $tinggi \times lebar\ NDRange$ yang digunakan oleh OpenCL *kernel* untuk matriks tersebut adalah $(32 \times 16) \times (16 \times 128/4) = 512 \times 512$. Kedua matriks tersebut memunculkan ukuran *NDRange* yang sama, sehingga kecepatan eksekusi OpenCL *kernel* seharusnya sama. Hal ini dapat dilihat pada Tabel 4.14. Terlihat bahwa OpenCL *kernel* memiliki kecepatan yang kurang lebih sama pada kedua jenis matriks masukan, sedangkan kecepatan *optimized kernel* jauh lebih baik ketika *batch* dan kanal matriks keluaran berukuran besar.

Tabel 4.12: Spesifikasi matriks pertama yang memiliki tinggi dan lebar matriks masukan yang berukuran besar.

Matriks	**Kedalaman**	**Tinggi**	**Lebar**	**Banyak Batch**
Image	4	516	516	1
Filter	4	5	5	4
Output	4	512	512	1

Tabel 4.13: Spesifikasi matriks kedua yang memiliki *batch* dan kanal matriks keluaran yang banyak.

Matriks	**Kedalaman**	**Tinggi**	**Lebar**	**Banyak Batch**
Image	4	20	20	32
Filter	4	5	5	128
Output	128	16	16	32

Tabel 4.14: Hasil perbandingan kecepatan Tensorflow Lite *kernel* pada dua jenis matriks masukan, dimana nilai-nilai pada tabel adalah rata-rata dari 10 kali eksekusi dalam milidetik.

Ukuran Keluaran	**Naive Kernel**	**Optimized Kernel**	**OpenCL Kernel**	**OpenCL + Transfer Data**
$Tinggi \times Lebar = 512 \times 512$	1300.6688	431.8264	79.6208	107.6360
$BanyakBatch \times Kedalaman = 32 \times 128$	1301.6718	55.8074	79.6087	96.9623

Dalam penelitian ini, *optimized kernel* hanya digunakan sebagai pembanding terhadap OpenCL *kernel* yang telah diimplementasikan oleh penulis. Penulis tidak mempelajari algoritma yang digunakan dalam implementasi *optimized kernel* secara mendalam. Analisis terhadap alasan mengapa *optimized kernel* memiliki kecepatan yang lebih baik saat *batch* dan kanal matriks keluaran berukuran besar daripada saat tinggi dan lebar matriks masukan berukuran besar berada diluar lingkup penelitian. Namun, menurut penulis alasan utama kecepatan OpenCL *kernel* tidak dapat mengungguli kecepatan *optimized kernel* pada beberapa kasus adalah karena operasi baca/tulis terhadap memori GPU hampir selalu menjadi *bottleneck* dalam suatu OpenCL *kernel* [15]. Komputasi bukanlah penyebab lambatnya suatu OpenCL *kernel* karena telah dijelaskan bahwa GPU memiliki performa komputasi yang sangat tinggi.

Pada semua kasus pengujian, dapat terlihat bahwa transfer data antara memori CPU dan GPU sangat mempengaruhi kecepatan OpenCL *kernel* pada matriks-matriks yang berukuran kecil. Perhatikan bahwa pada semua kasus uji, transfer data antara memori GPU dan CPU menjadi *bottleneck* ketika matriks berukuran kecil. Pada tabel hasil eksperimen, waktu yang diperlukan untuk transfer data antara memori CPU dan GPU adalah selisih antara waktu eksekusi OpenCL *kernel* saja dengan waktu eksekusi OpenCL *kernel* ditambah transfer data antara memori CPU dan GPU (selisih antara kolom keempat dan kolom kelima). Dapat dilihat bahwa selisih tersebut lebih besar daripada waktu eksekusi OpenCL *kernel* saja ketika matriks-matriks yang terlibat berukuran relatif kecil.

Pada matriks-matriks besar, transfer data antara memori CPU dan GPU tidak menjadi *bottleneck*. Hal ini disebabkan oleh dua hal. Pertama, beban komputasi lebih berat ketika matriks berukuran besar. Kedua, beban untuk operasi baca/tulis terhadap memori GPU lebih juga berat ketika ukuran matriks semakin besar. Telah dijelaskan sebelumnya bahwa operasi baca/tulis terhadap memori GPU sering menjadi *bottleneck* dari suatu OpenCL *kernel*. Operasi baca/tulis tersebut memiliki peran lebih besar terhadap meningkatnya waktu eksekusi OpenCL *kernel*. Jadi, operasi baca/tulis terhadap memori GPU pada OpenCL *kernel* memiliki peran yang besar terhadap berpindahnya *bottleneck* dari transfer data antar memori ke eksekusi OpenCL *kernel*.

BAB 5
KESIMPULAN DAN SARAN

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

5.1 Kesimpulan

OpenCL dapat digunakan untuk menjalankan *Deep Learning inference* pada perangkat *mobile*. Melalui OpenCL, operasi-operasi matriks pada *Deep Learning infernce* dapat dijalankan di *mobile* GPU. Untuk melakukannya, diperlukan pustaka dan *headers* dari OpenCL yang dapat diperoleh dari masing-masing vendor GPU. Implementasi operasi-operasi *Deep Learning inference* menggunakan OpenCL dapat dilakukan pada Tensorflow Lite yang merupakan pustaka *Deep Learning* bersifat sumber terbuka. Implementasi operasi-operasi *Deep Learning inference* menggunakan OpenCL dapat ditambahkan sebagai *kernel* baru pada Tensorflow Lite. Dengan demikian, pengguna dapat memilih untuk menggunakan OpenCL *kernel* ketika melakukan kompilasi Tensorflow Lite. Pada penelitian ini penulis telah menambahkan OpenCL *kernel* pada Tensorflow Lite untuk operasi perkalian matriks-matriks dan konvolusi matriks yang berjalan di GPU.

Pengujian telah dilakukan terhadap OpenCL *kernel* yang telah diimplementasikan. Hasil pengujian menunjukkan bahwa OpenCL *kernel* yang berjalan di GPU untuk operasi perkalian matriks-matriks memiliki kecepatan yang lebih baik daripada *naive kernel* dan *optimized kernel* dari Tensorflow Lite yang berjalan di CPU. Keunggulan kecepatan dari OpenCL kernel semakin terlihat ketika ukuran matriks-matriks masukan semakin besar. Sementara itu pada operasi konvolusi matriks, OpenCL *kernel* memiliki kecepatan yang lebih baik daripada dua *kernel* lain ketika banyaknya *batch* dan kanal dari matriks keluaran relatif kecil. Saat banyaknya *batch* dan kanal matriks keluaran cukup besar, kecepatan dari OpenCL *kernel* tidak lebih baik daripada *optimized kernel* yang berjalan di CPU.

Pada dua operasi matriks yang telah diuji, diketahui bahwa persiapan pada OpenCL yang berupa proses penyalinan data antar memori CPU dan GPU cukup berpengaruh terhadap kecepatan OpenCL *kernel*. Pada matriks-matriks yang relatif kecil, proses menyalin data ini menjadi *bottleneck* dari OpenCL *kernel*. Persiapan OpenCL tersebut berkontribusi lebih dari 50 persen dari total waktu eksekusi OpenCL *kernel* pada matriks kecil. *Bottleneck* pada persiapan OpenCL tersebut

semakin menghilang ketika ukuran matriks masukan semakin besar. Pada matriks-matriks yang cukup besar, waktu yang diperlukan untuk eksekusi *kernel* jauh lebih besar daripada waktu yang diperlukan untuk persiapan OpenCL. Salah satu penyebabnya adalah semakin banyaknya operasi baca/tulis terhadap memori GPU yang dilakukan oleh *kernel* ketika ukuran matriks semakin besar.

5.2 Saran

Pada penelitian ini penulis telah mengimplementasikan program OpenCL untuk operasi perkalian matriks-matriks dan konvolusi matriks. Batasan pada penelitian ini adalah operasi konvolusi matriks yang telah diimplementasikan menggunakan OpenCL hanya dapat digunakan ketika besarnya jangkah adalah satu. Ukuran jangkah yang lebih besar dari satu menyebabkan blok matriks masukan yang diproses oleh suatu *work-group* menjadi lebih besar. Jika blok terlalu besar, *local memory caching* tidak dapat dilakukan karena keterbatasan kapasitas memori lokal. Bahkan ketika besarnya jangkah sama dengan satu, tidak semua GPU dapat menjalankan OpenCL *kernel* akibat keterbatasan memori. Untuk penelitian selanjutnya, disarankan membuat implementasi OpenCL untuk operasi konvolusi yang mendukung semua ukuran jangkah, namun tetap memiliki kecepatan yang baik. Untuk melakukannya mungkin diperlukan pendekatan lain selain *local memory caching* agar kapasitas memori tidak menjadi masalah.

Program OpenCL pada penelitian ini hanya diimplementasikan untuk perangkat Android. Selain untuk Android, Tensorflow Lite dan OpenCL sebenarnya juga dapat dijalankan pada perangkat IoS. Penulis menyarankan percobaan penerapan OpenCL untuk *Deep Learning inference* pada perankat IoS. Perangkat IoS memiliki dukungan yang lebih baik terhadap OpenCL daripada perangkat Android. Penulis melihat kemungkinan bahwa *driver* dari OpenCL pada perangkat IoS memiliki kecepatan yang lebih baik daripada Android. Untuk perangkat Android sendiri, eksekusi *Deep Learning inference* di GPU sebenarnya tidak harus menggunakan OpenCL. Terdapat pilihan API lain seperti Vulkan [17] dan OpenGL [16] untuk melakukan pemrograman paralel di GPU. Penulis telah mengimplementasikan Tensorflow Lite *kernel* menggunakan Vulkan dengan pendekatan yang sama untuk operasi perkalian matriks-matriks dan konvolusi matriks. Meskipun pendekatannya sama, kecepatan yang dihasilkan kurang memuaskan. Pada beberapa kasus, Vulkan *kernel* bahkan tidak dapat mengungguli *naive kernel*. Pada penelitian selanjutnya disarankan penggunaan Vulkan untuk mengimplementasikan operas-operasi *Deep Learning inference* namun dengan pendekatan yang berbeda agar kecepatan-

nya maksimal.

Keberhasilan implementasi OpenCL untuk operasi-opearasi *Deep Learning inference* yang berjalan di GPU memunculkan kemungkinan untuk penggunaan berbagai jenis prosesor lain dalam menjalankan *Deep Learning inference*. Terdapat perangkat-perangkat yang memiliki dukungan dari prosesor selain CPU dan GPU, misalnya DSP dan FPGA. Pada penelitian selanjutnya disarankan untuk meneliti bagaimana kecepatan *Deep Learning inference* pada perangkat yang memiliki dukungan DSP atau FPGA. OpenCL dapat kembali dimanfaatkan untuk melakukan pemrograman pada prosesor-prosesor tersebut. Program OpenCL untuk operasi perkalian matriks-matriks dan konvolusi matriks yang telah diimplementasikan pada penelitian ini juga dapat digunakan ulang untuk prosesor-prosesor lain dengan beberapa penyesuaian.

DAFTAR REFERENSI

[1] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. 2011. OpenCL Programming Guide (1st ed.). Addison-Wesley Professional.

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

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

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

[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] Sepp Hochreiter and Jrgen Schmidhuber. 1997. Long Short-Term Memory. Neural Comput. 9, 8 (November 1997), 1735-1780. DOI=http://dx.doi.org/10.1162/neco.1997.9.8.1735

[14] 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

[15] Wang, H., Yun, J., & Bourd, A. (2018). OpenCL Optimization and Best Practices for Qualcomm Adreno GPUs. In Proceedings of the International Workshop on OpenCL - IWOCL 18. ACM Press. https://doi.org/10.1145/3204919.3204935

[16] https://www.khronos.org/opengl/

[17] https://www.khronos.org/vulkan/

[18] Y. Le Cun, B. Boser, J. S. Denker, R. E. Howard, W. Habbard, L. D. Jackel, and D. Henderson. 1990. Handwritten digit recognition with a back-propagation network. In Advances in neural information processing systems 2, David S. Touretzky (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA 396-404.

LAMPIRAN

LAMPIRAN 1