arm AI

Optimizing NN inference performance on Arm NEON and Vulkan using ailia SDK

ax Inc

David Cochard & Kazuki Kyakuno

21st September 2021

© 2021 Arm

arm

Welcome!

Tweet us: @ArmSoftwareDev -> #AIVTT

Check out our Arm Software Developers YouTube channel

Signup now for our next AI Virtual Tech Talk: developer.arm.com/techtalks

Attendees: don’t forget to fill out the survey to be in with a chance of winning an Arduino Nano 33 BLE board

© 2020 Arm Limited (or its affiliates)

Our upcoming Arm AI Tech Talks

Date	Title	Host
September 21st	Optimizing NN inference performance on Arm NEON and Vulkan using the Ailia SDK	Ax Inc
October 5th	EON Tuner: AutoML for real-world embedded devices	Edge Impulse
October 28th	ARM架构端侧AI视觉算法的开发到部署 (Development to Deployment of Endpoint AI vision Algorithms Based on Arm Architecture)	ICE TECH
November 2nd	Getting started with running Machine Learning on Arm Ethos-U55	Arm

Visit: developer.arm.com/techtalks

Presenters

David Cochard,
Engineering Manager,
ax Inc.

Kazuki Kyakuno,
CTO,
ax Inc.

Agenda

- Presentation of our solution “ ailia ”
- Optimize computation on CPU
- Optimize computation on GPU

Company Profile

Name	ax Inc.
Location	Tokyo, Japan
CEO	TERADA Takehiko
Business	・Development and provide “ailia SDK” ・AI Consulting, Training AI models and more AI businesses

"AI everywhere "

We believe that AI will be used on every device in the near future.
We are developing fast SDKs and constantly researching the latest AI models
to help accelerating the evolution of AI era.

ax Inc. aims to provide tools to implement the latest AI capabilities to solve
real-world problems.

There are **many challenges**
in implementing AI for various devices.

Those challenges include:

- Wide variety of AI models
- Multi platform
- Various programing languages
- Long-term API consistency
- Performance optimization for each devices

and more.

ailia SDK

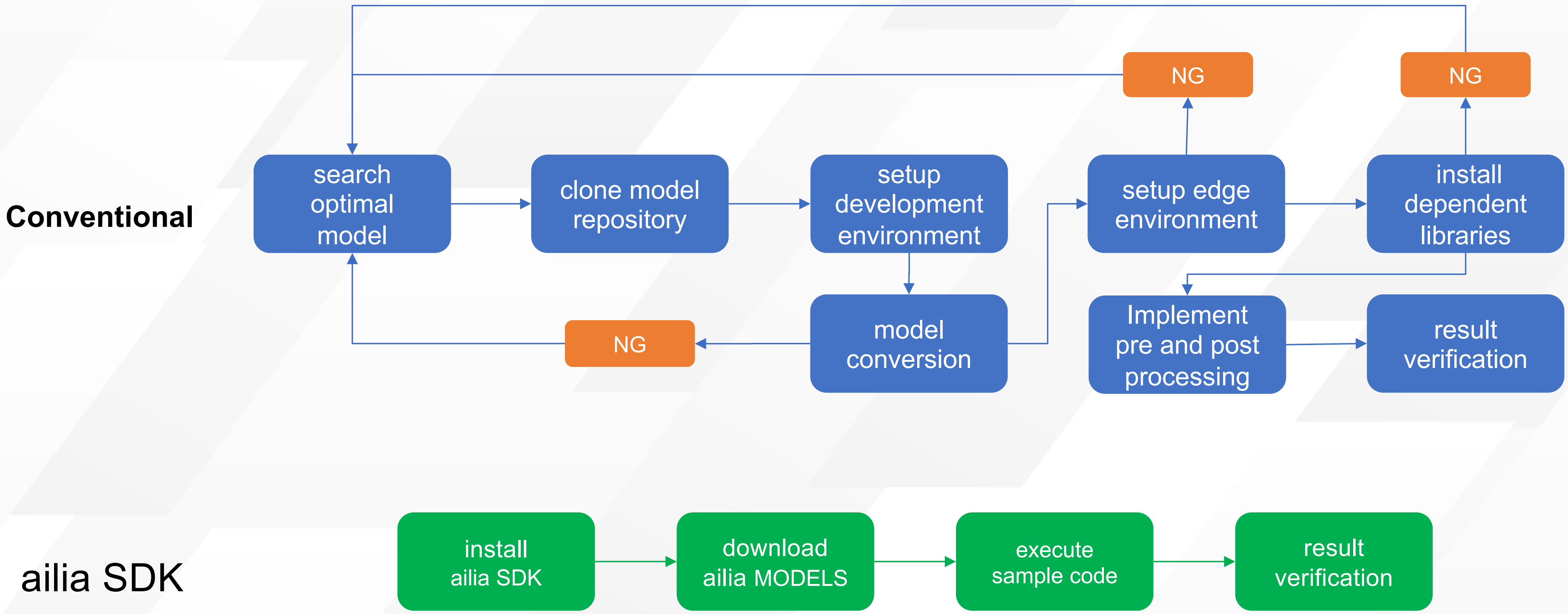
ailia SDK is an AI framework leveraging CPU and GPU to achieve high-performance AI inference

It supports ONNX (opset 10 & 11) and enables high-performance inference using NEON and Vulkan

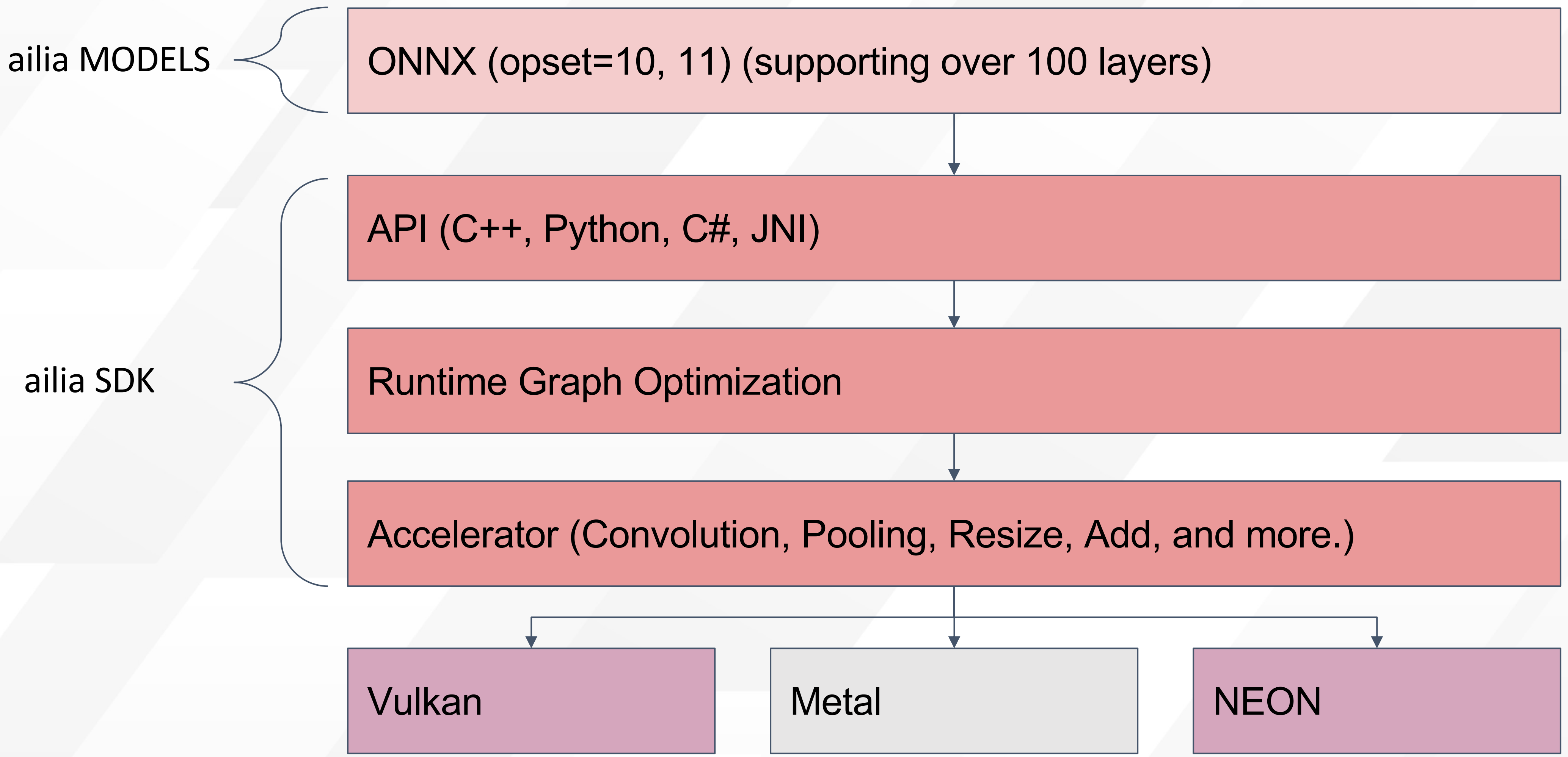
It offers over 140 pre-trained models, ready to be integrated into our client’s application

https://ailia.jp/en

Benefits when implementing AI to Edge devices



ailia SDK Architecture

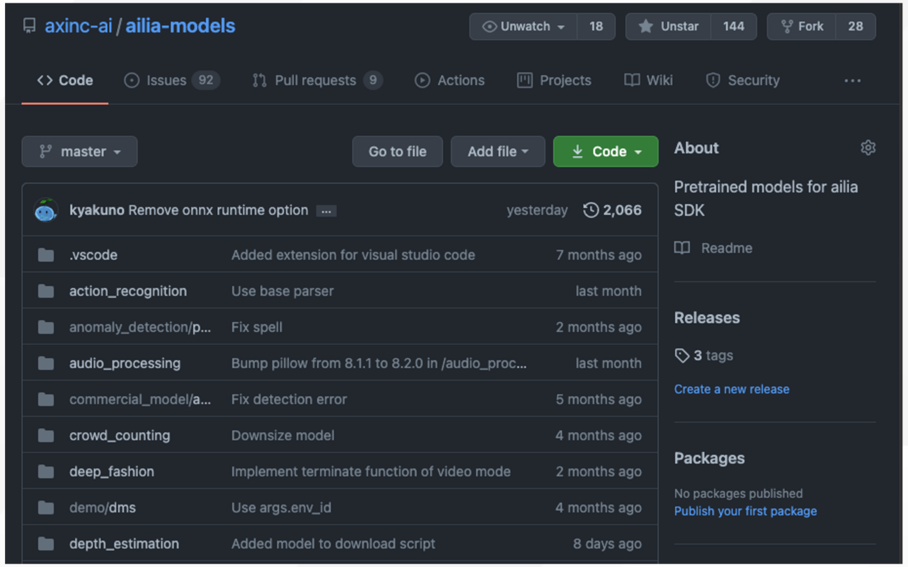


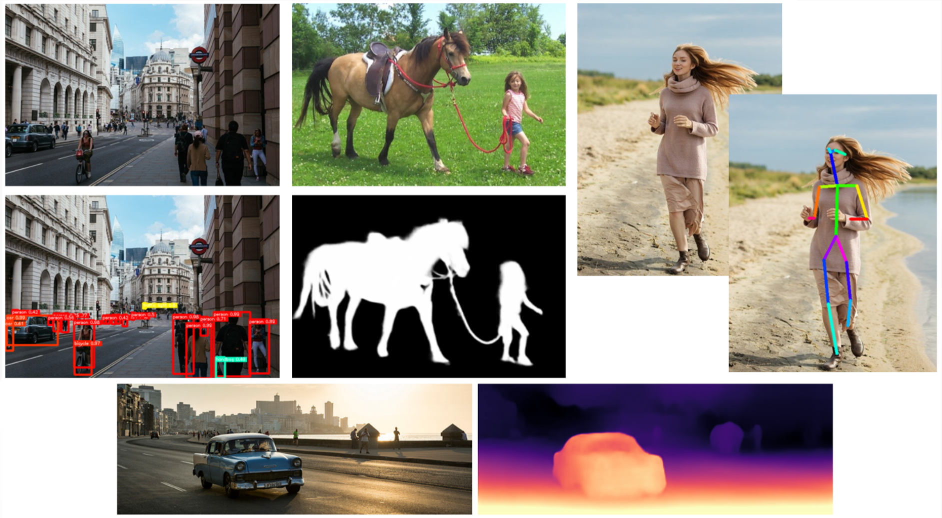
ailia MODELS

Over 140 models compatible with ailia SDK are publicly available on github

You can easily try out the latest models such as YOLOv4, MIDAS and PaddleOCR

https://github.com/axinc-ai/ailia-models

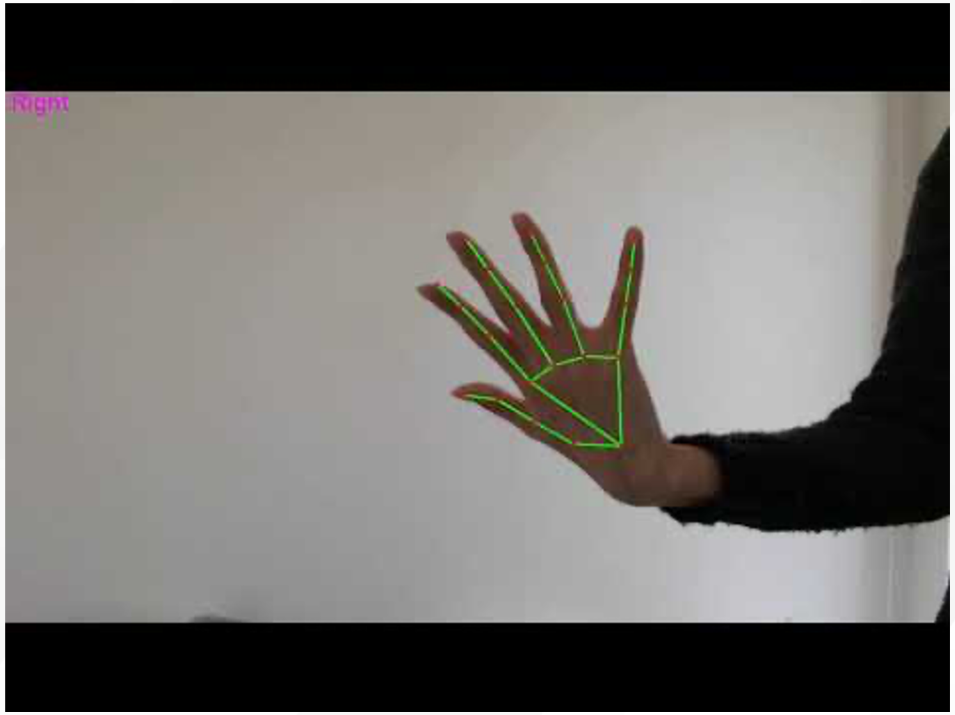




ailia MODELS

ailia MODELS has samples for Python, C ++ and Unity.

You can use accelerated inference using Vulkan in any environment.



Hand Detection

Depth Estimation

Optimize computation on CPU

NEON implementation in AI

Since the amount of processing is large for AI compared to general applications, significant speedup is required.

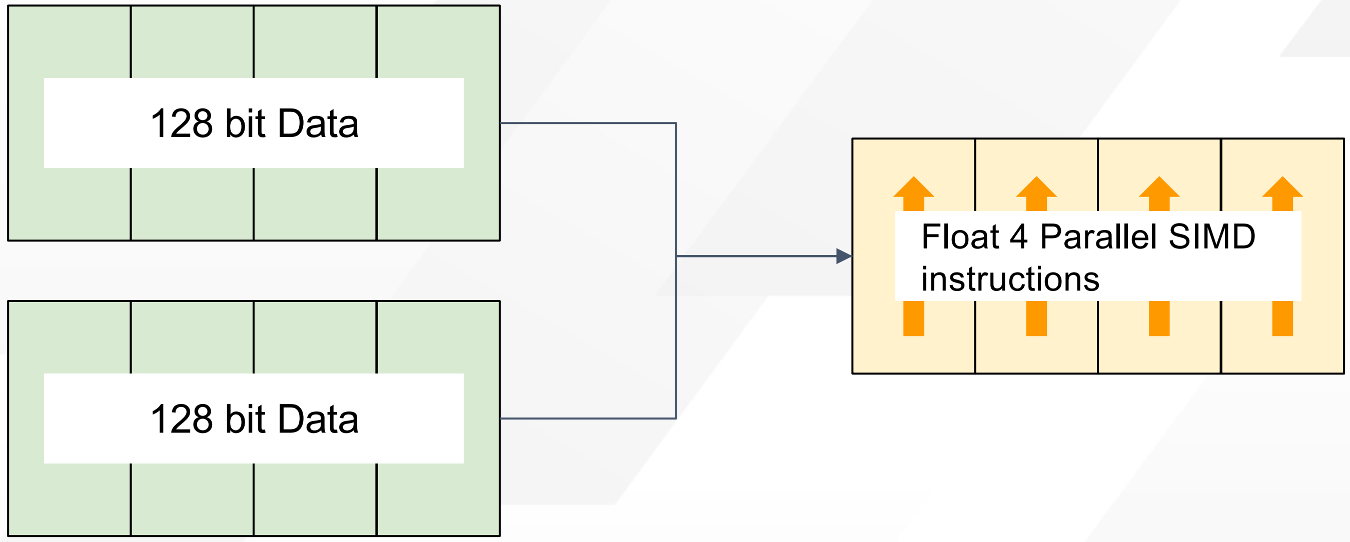
Thanks to the high degree of parallelism in layer operations, there is a lot of room for optimization using SIMD instructions such as NEON.

There are many types of layers to implement and possibilities to use various optimization techniques.

NEON overview

NEON provides scalar/vector instructions and registers for Arm CPUs

It is possible to perform parallel calculation in 128-bit units, as well as FP32, up to 4 elements simultaneously



Basic techniques

Replace code branching with a bit select instruction

Process number sign using by bit manipulation

Reorder data structure (load / store)

```
// remove branch with bit select
// dst[i]=(src[i]<0.0f)?src[i]*alpha:src[i];
val = vld1q_f32(src);
sel = vcltq_f32(val, zero);
mod = vmulq_n_f32(val, alpha);
vst1q_f32(dst, vbslq_f32(sel, mod, val));
```

```
// reorder data with structure load
x[] = {
  11, 12, 13, 14,
  21, 22, 23, 24,
  31, 32, 33, 34,
  41, 42, 43, 44,
};
v = vld4q_f32(x);
// v.val[0] : { 11, 21, 31, 41 }
// v.val[1] : { 12, 22, 32, 42 }
// v.val[2] : { 13, 23, 33, 43 }
// v.val[3] : { 14, 24, 34, 44 }
```

```
// save sign bit from input value.
mask = vdupq_n_u32(1<<31);
sign = vandq_u32(cast_u32(x), mask);

v = vabsq_f32(x);
// .. any operation with abs value ..

// restore sign bit with xor
res = cast_f32(veorq_u32(cast_u32(v), sign));
```

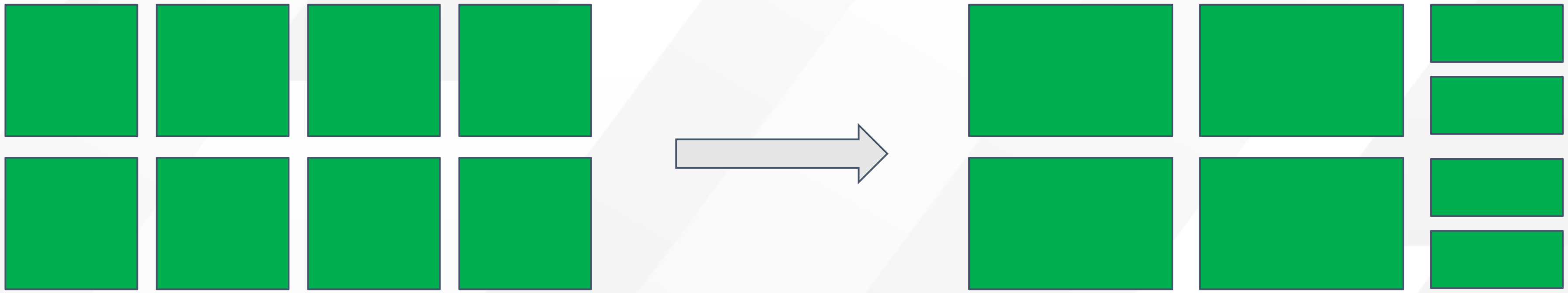
Approximation

NEON implementation using approximate expressions for high-level functions such as exp, log, and erf

```
// log(x) = log((2^n) * z) = n*log(2) + log(z)
// log(z) ≒ 2 * (w + (w^3)/3 + (w^5)/5 + ..) : w = (z-1)/(z+1)
log2 = 0.6931471805599453f;
n = pick_exponent(x);
z = pick_fractional(x);
w = (z-1.0) / (z+1.0);
ww = w*w;
r = (1.0/7.0) + (1.0/9.0)*ww;
r = (1.0/5.0) + r*ww;
r = (1.0/3.0) + r*ww;
r = (1.0 + r*ww);
r = r*w; // w + (w^3)/3 + (w^5)/5 + (w^7)/7 + (w^9)/9
return (n*log2 + r*2);
```

Threading

Taking advantage of Arm big.little technology, performance gain can be achieved by efficiently assigning jobs to different cores.



Diving a task in units of equal size does not fit big.little SoC. design

Assign more processing-intensive tasks to big cores, and smaller ones to little cores.

Adjust the processing unit according to the cache size

Benchmark

Comparison of inference time with NEON enabled and disabled

SoC	Model	Improvement (w/wo NEON)
Snapdragon 888	ResNet50	2.15 times faster
	YOLOv3	3.90 times faster
Exynos 9820	ResNet50	3.08 times faster
	YOLOv3	2.86 times faster

Future work

The vector length of NEON is 128 bit, but 512 bit vector operation can be used in SVE2 added in Armv9-A

Going forward, ailia SDK will continue to support the latest instruction sets

Optimize computation on GPU

Benefits of Vulkan

Support for all major GPUs

Support for all major OSs

Windows, Android, Linux

Easy installation for GPU inference

Being widely used for gaming, it only requires standard drivers to run

Little additional disk space usage

ailia_vulkan.dll is only 2.8MB

AI Inference Acceleration with Vulkan

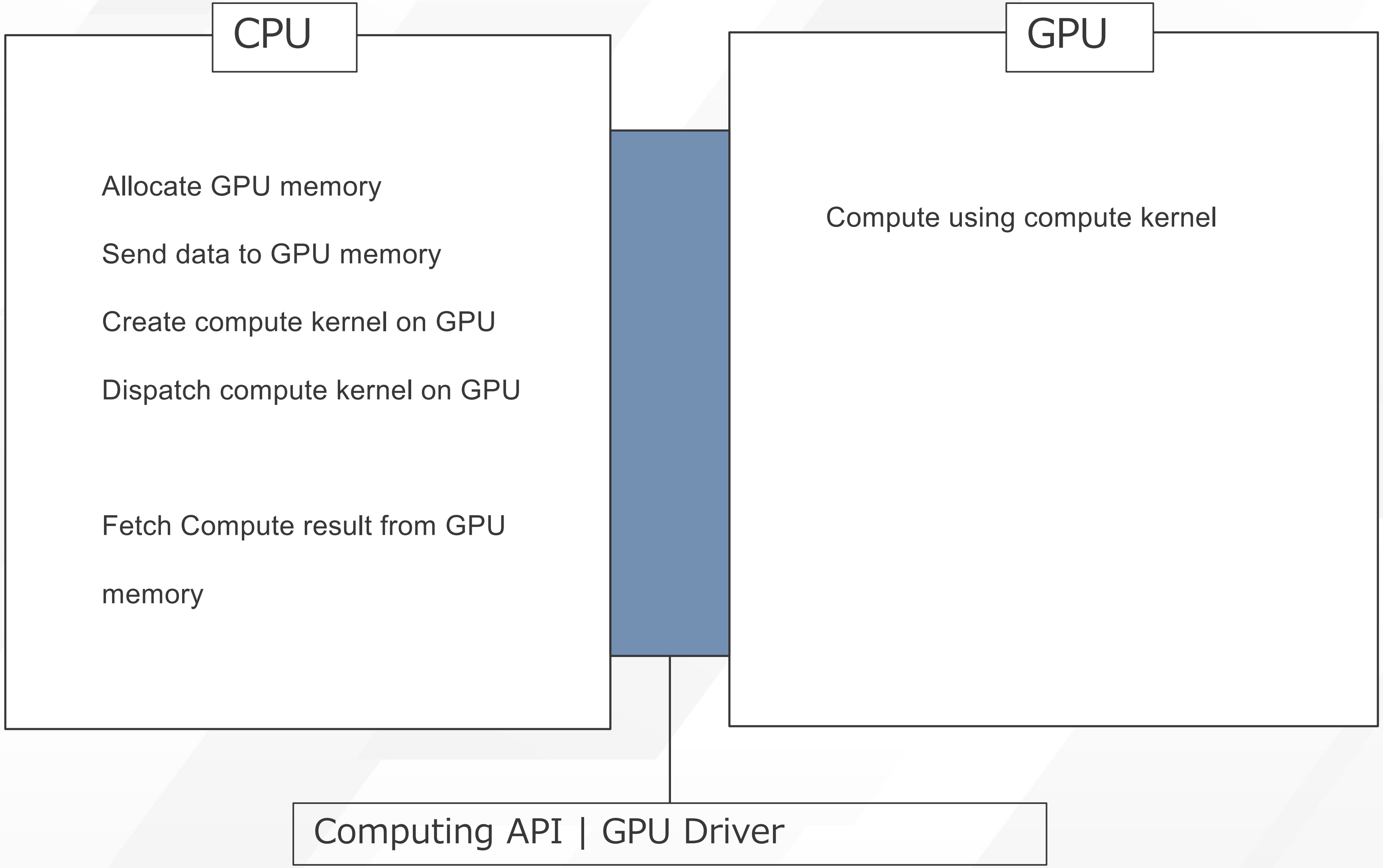
Fast AI inference achieved using Runtime Graph Optimization and Layer Fusion

Implementation of the Optimized Logic for GEMM using Vulkan’s Compute Shader

Implementation of layers such as Convolution or Pooling using Vulkan’s Compute Shader

Implementation of the Winograd Algorithm with shaders to accelerate the heavy Convolution layer

Roles between CPU and GPU



Code required for Vulkan

Create VkInstance

Select VkPhysicalDevice

Create VkDevice

Get VkQueue

Allocate VkDeviceMemory

Bind vkDeviceMemory to VkBuffer

Send data to device from host

Configure VkShaderModule

Create VkDescriptorSetLayout

Create VkPipelineLayout

Create VkPipeline

Create VkDescriptorPool

Create VkDescriptorSet

Create VkCommandPool

Create VkCommandBuffer

Regist VkPipeline to VkCommandBuffer

Regist VkDescriptorSet to VkCommandBuffer

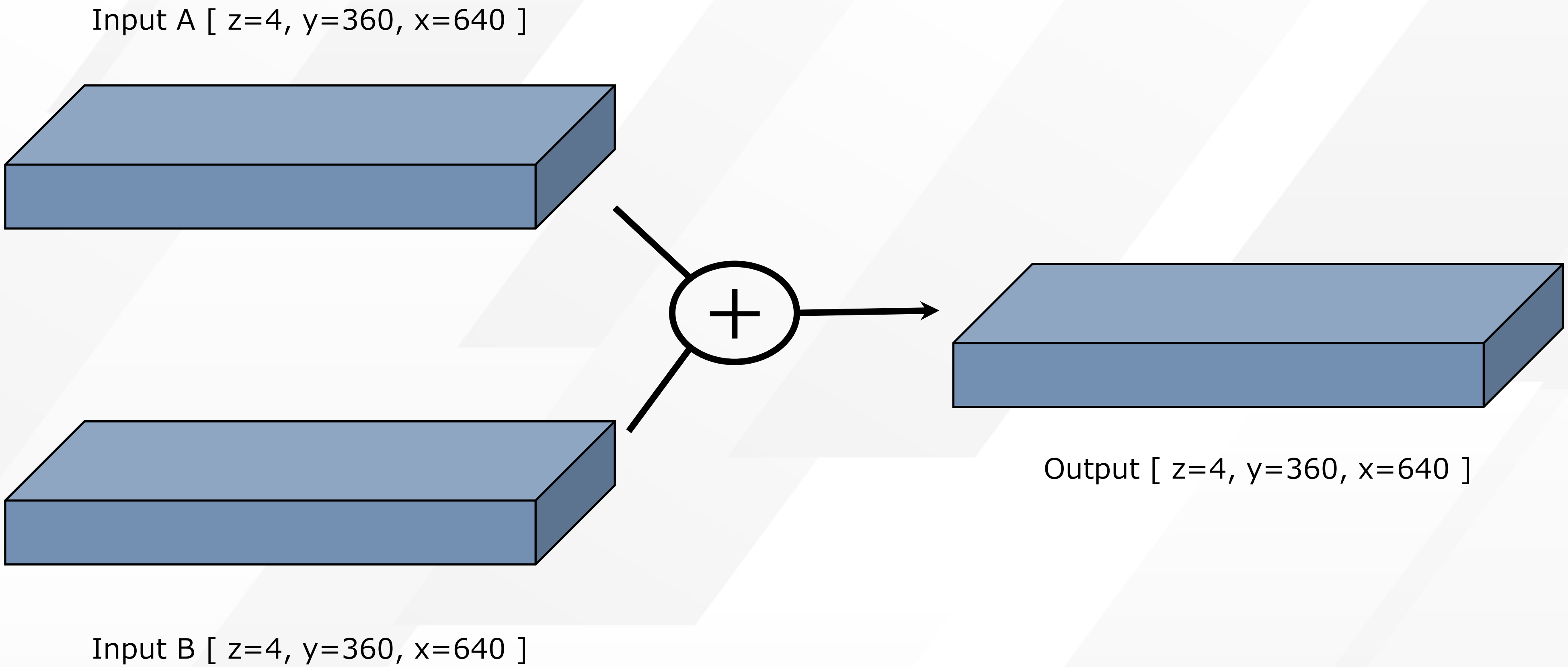
Call vkCmdDispatch

Executing and waiting for VkCommandBuffer to complete

Data transfer from device to host

Example of GPU Computing

Add the corresponding elements of A and B of the same size and write to the corresponding element of the output of the same size



Example of GPU Computing : Device code (GPU code)

```
#version 450
layout(std430, binding = 0) writeonly buffer Dst {
    float data[];
} dst;
layout(std430, binding = 1) readonly buffer Src_A {
    float data[];
} src_a;
layout(std430, binding = 2) readonly buffer Src_B {
    float data[];
} src_b;

layout(local_size_x = 64) in;
void main()
{
    // Exception handling of fractional blocks is omitted
    const uint id = gl_GlobalInvocationID.x;
    dst.data[id] = src_a.data[id] + src_b.data[id];
}
```

Example of GPU Computing : Host code (CPU code)

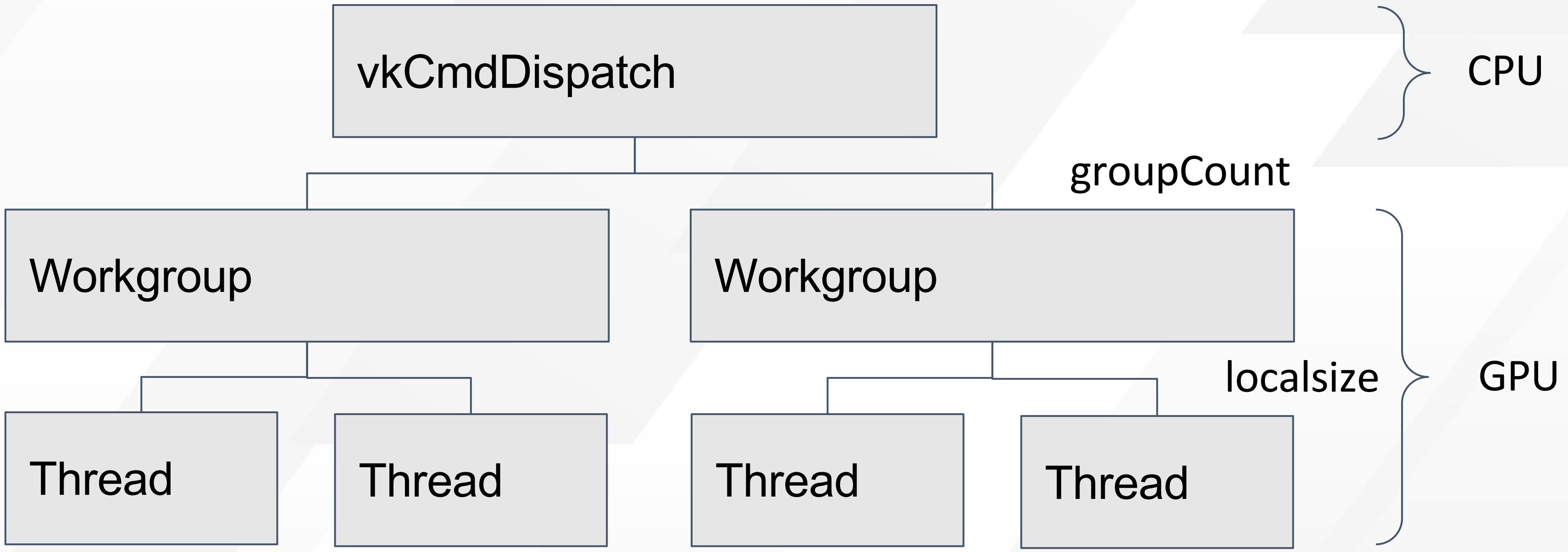
```
// Instance initialization, device selection, buffer allocation, shader module construction, etc. are
omitted.
vkBeginCommandBuffer(cmdBuf, &beginInfo); // Start recording command buffer

// Registered pipeline in command buffer --Shader module is registered in pipeline
vkCmdBindPipeline(cmdBuf, VK_PIPELINE_BIND_POINT_COMPUTE, pipeline);

// Registered descriptor set in command buffer --I / O buffer allocation is registered in descSet
vkCmdBindDescriptorSets(cmdBuf, VK_PIPELINE_BIND_POINT_COMPUTE, descSet);

// Specify how many work loops to start
// Start with the most recently configured pipeline and descriptor set
// Numerical example of one-dimensional processing of z = 4, y = 360, x = 640 with localsize = 64
vkCmdDispatch(cmdBuf, (4*360*640)/64, 1, 1);

vkEndCommandBuffer(cmdBuf); // Set the end of command buffer construction

// Pass the command buffer to the device and run the shader --cmdBuf is registered in submitInfo
vkQueueSubmit(queue, 1, &submitInfo, NULL);

// Waiting for processing completion and data collection are omitted
```

Workgroups and Threads (Invocation)

Write how many kernels to boot in the host (CPU) code (groupCount)

Describe what each thread does in the device (GPU) code (shader)

Use localsize to specify the number of threads (invocations)



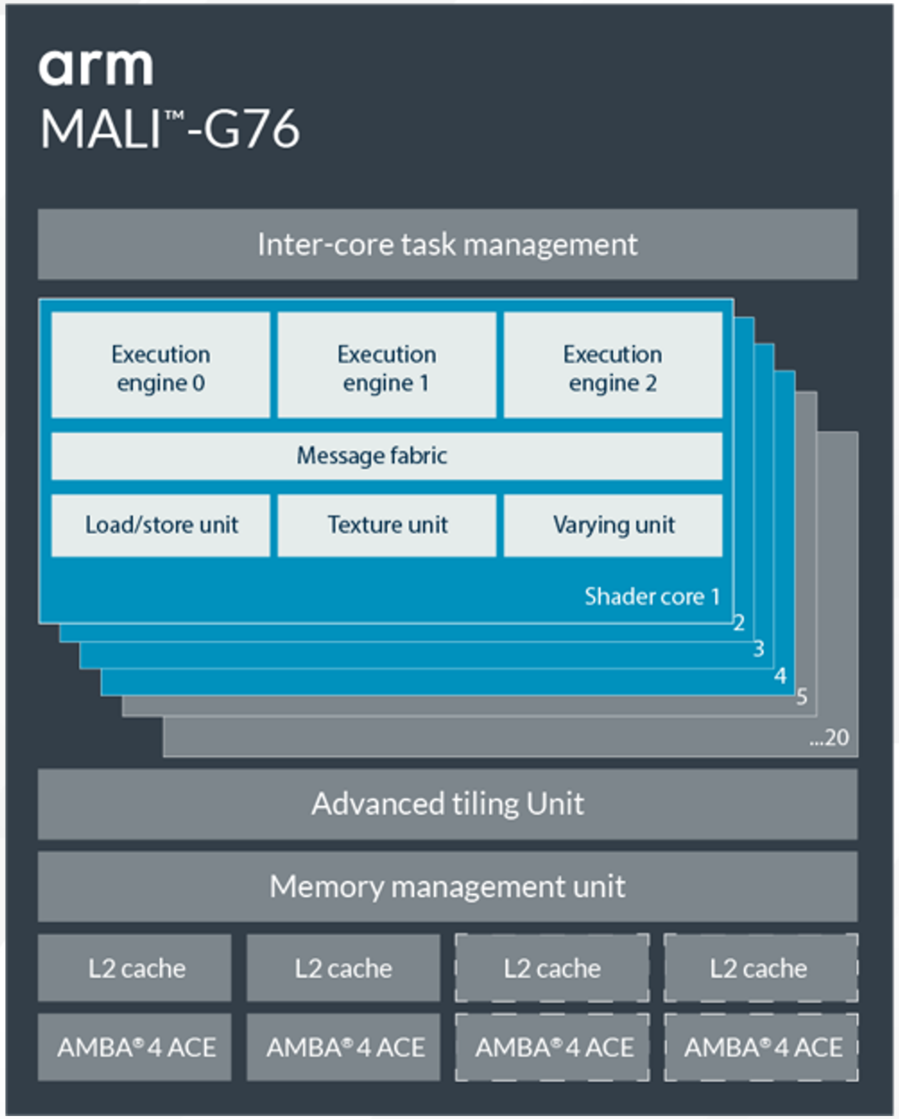
Arm GPU HW configuration example

Configurable from 4 to 20 shader cores

delivering largest capability for a Mali GPU

3 engines per shader core

8 execution lanes per engine



https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus/mali-g76-gpu

Correspondence with API

vkCmdDispatch(.., groupCountX, groupCountY, groupCountZ);

Kernel boot process with host code

Launch (groupCountX * groupCountY * groupCountZ) workgroups

One workgroup is assigned to one execution engine

If you can fill all the execution engines, make full use of the GPU

layout(localsize_x=64) in;

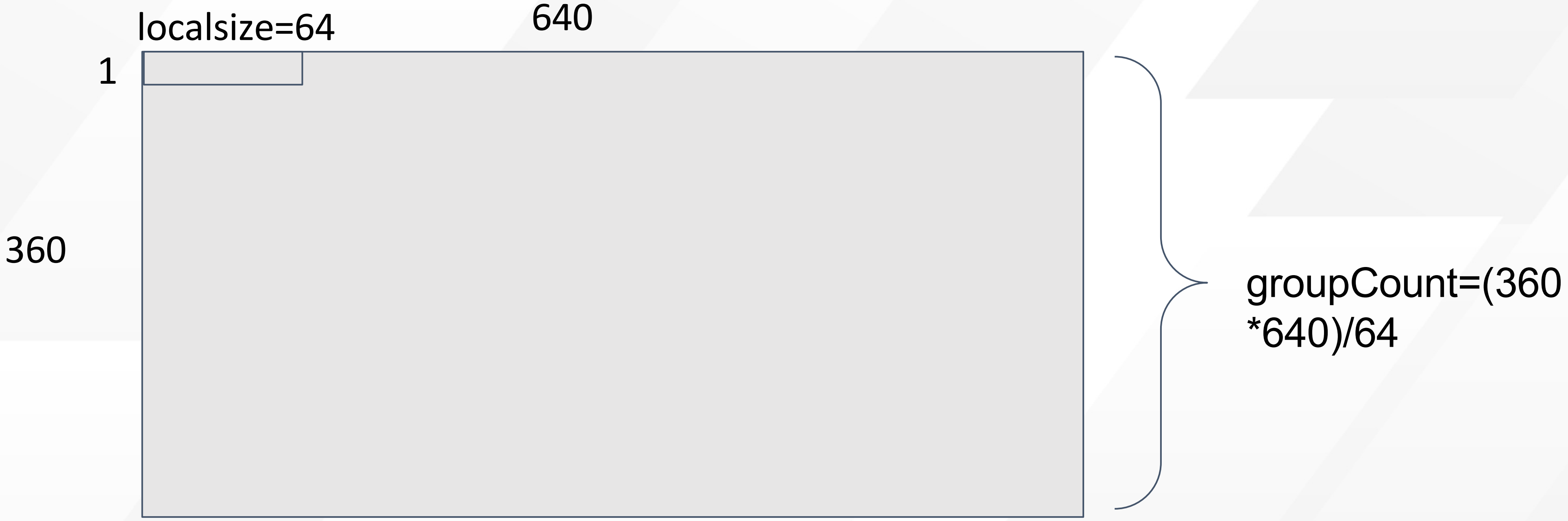
Device code thread number specification part

Specifies the number of local threads in a workgroup

If the number of threads issued in the execution engine can be filled, the execution engine will be fully utilized.

Relation between localsize and dispatch

If the local size is 64 and the image size is 640 * 360, you need to run (360 * 640) / 64 workgroups to run the entire image



How to choose localsize

The limit of localsize is defined by maxComputeWorkgroupSize

maxComputeWorkgroupSize = {384,384,384} (Mali G76)

maximum size of a local compute workgroup per dimension

maxComputeWorkGroupInvocations = 384 (Mali G76)

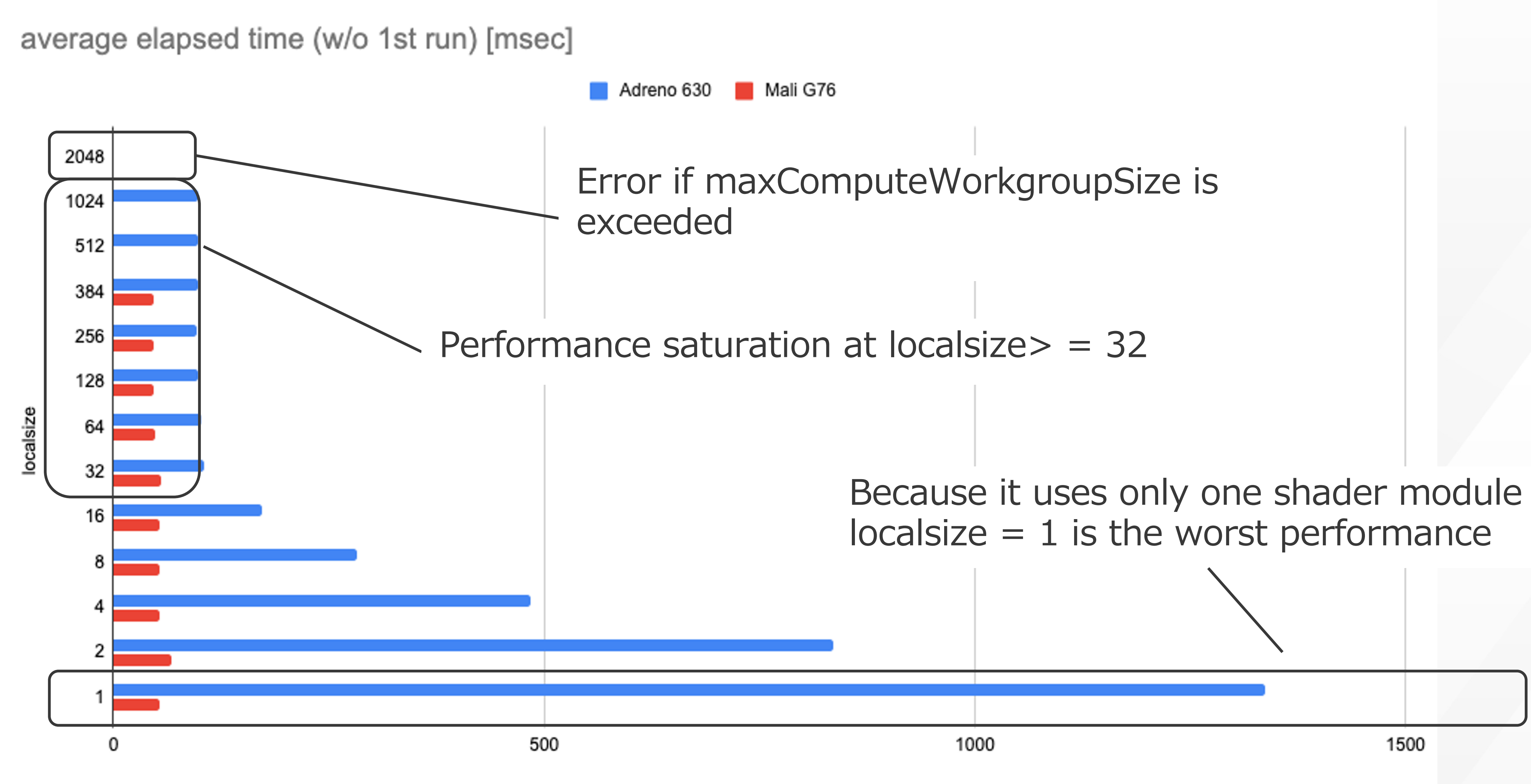
maximum total number of compute shader invocations in a single local workgroup

Arm Mali GPU Best Practices Developer Guide said

"Use 64 as a baseline workgroup size. Do not use more than 64 threads per workgroup."

Localsize adjustment required for complex and heavy kernels

Time spent by localsize



GEMM

General matrix multiply

Multiplication of 2D dense matrix

Often used in scientific computing (computer simulation)

Patterson & Hennessy "Computer Organization and Design"

1426-page textbooks with the story of speeding up GEMM

Basic implementation of GEMM

```
// I / O definition omitted (when trans_a == false && trans_b == false)
#define BLOCK_SIZE 8
layout(local_size_x=BLOCK_SIZE, local_size_y=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared float sa[ BLOCK_SIZE * BLOCK_SIZE ];
shared float sb[ BLOCK_SIZE * BLOCK_SIZE ];
void main()
{
    uint lx = gl_LocalInvocationID.x; uint ly = gl_LocalInvocationID.y;
    uint dx = gl_WorkGroupID.x * BLOCK_SIZE; uint dy = gl_WorkGroupID.y * BLOCK_SIZE;
    float sum = 0.0;
    for (uint k=0; k<TOTAL_K; k+=BLOCK_SIZE) {
        sa[ ly * BLOCK_SZIE + lx ] = src_a.data[ (dy + ly) * src_a_width + ( k+lx) ];
        sb[ ly * BLOCK_SIZE + lx ] = src_b.data[ ( k + ly) * src_b_width + (dx+lx) ];
        barrier();
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum += sa[ ly * BLOCK_SIZE + i ] * sb[ i * BLOCK_SIZE + lx ];
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    dst.data[ (dy+ly) * dst_width + (dx+lx) ] = sum;
}
```

Basic implementation of GEMM

```
// I / O definition omitted (when trans_a == false && trans_b == false)
#define BLOCK_SIZE 8
layout(local_size_x=BLOCK_SIZE, local_size_y=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared float sa[ BLOCK_SIZE * BLOCK_SIZE ];
shared float sb[ BLOCK_SIZE * BLOCK_SIZE ];
void main()
{
    uint lx = gl_LocalInvocationID.x; uint ly = gl_LocalInvocationID.y;
    uint dx = gl_WorkGroupID.x * BLOCK_SIZE; uint dy = gl_WorkGroupID.y * BLOCK_SIZE;
    float sum = 0.0;
    for (uint k=0; k<TOTAL_K; k+=BLOCK_SIZE) {
        sa[ ly * BLOCK_SZIE + lx ] = src_a.data[ (dy + ly) * src_a_width + ( k+lx) ];
        sb[ ly * BLOCK_SIZE + lx ] = src_b.data[ ( k + ly) * src_b_width + (dx+lx) ];
        barrier();
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum += sa[ ly * BLOCK_SIZE + i ] * sb[ i * BLOCK_SIZE + lx ];
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    dst.data[ (dy+ly) * dst_width + (dx+lx) ] = sum;
}
```

1 thread is responsible for one output element

Basic implementation of GEMM

```
// I / O definition omitted (when trans_a == false && trans_b == false)
#define BLOCK_SIZE 8
layout(local_size_x=BLOCK_SIZE, local_size_y=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared float sa[ BLOCK_SIZE * BLOCK_SIZE ];
shared float sb[ BLOCK_SIZE * BLOCK_SIZE ];
void main()
{
    uint lx = gl_LocalInvocationID.x; uint ly = gl_LocalInvocationID.y;
    uint dx = gl_WorkGroupID.x * BLOCK_SIZE; uint dy = gl_WorkGroupID.y * BLOCK_SIZE;
    float sum = 0.0;
    for (uint k=0; k<TOTAL_K; k+=BLOCK_SIZE) {
        sa[ ly * BLOCK_SZIE + lx ] = src_a.data[ (dy + ly) * src_a_width + ( k+lx) ];
        sb[ ly * BLOCK_SIZE + lx ] = src_b.data[ ( k + ly) * src_b_width + (dx+lx) ];
        barrier();
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum += sa[ ly * BLOCK_SIZE + i ] * sb[ i * BLOCK_SIZE + lx ];
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    dst.data[ (dy+ly) * dst_width + (dx+lx) ] = sum;
}
```

localsize is 8 x 8 = 64
Create threads with 2D blocks

Basic implementation of GEMM

```
// I / O definition omitted (when trans_a == false && trans_b == false)
#define BLOCK_SIZE 8
layout(local_size_x=BLOCK_SIZE, local_size_y=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared float sa[ BLOCK_SIZE * BLOCK_SIZE ];
shared float sb[ BLOCK_SIZE * BLOCK_SIZE ];
void main()
{
    uint lx = gl_LocalInvocationID.x; uint ly = gl_LocalInvocationID.y;
    uint dx = gl_WorkGroupID.x * BLOCK_SIZE; uint dy = gl_WorkGroupID.y * BLOCK_SIZE;
    float sum = 0.0;
    for (uint k=0; k<TOTAL_K; k+=BLOCK_SIZE) {
        sa[ ly * BLOCK_SZIE + lx ] = src_a.data[ (dy + ly) * src_a_width + ( k+lx) ];
        sb[ ly * BLOCK_SIZE + lx ] = src_b.data[ ( k + ly) * src_b_width + (dx+lx) ];
        barrier();
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum += sa[ ly * BLOCK_SIZE + i ] * sb[ i * BLOCK_SIZE + lx ];
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    dst.data[ (dy+ly) * dst_width + (dx+lx) ] = sum;
}
```

Both A and B read into shared memory

Basic implementation of GEMM

```
// I / O definition omitted (when trans_a == false && trans_b == false)
#define BLOCK_SIZE 8
layout(local_size_x=BLOCK_SIZE, local_size_y=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared float sa[ BLOCK_SIZE * BLOCK_SIZE ];
shared float sb[ BLOCK_SIZE * BLOCK_SIZE ];
void main()
{
    uint lx = gl_LocalInvocationID.x; uint ly = gl_LocalInvocationID.y;
    uint dx = gl_WorkGroupID.x * BLOCK_SIZE; uint dy = gl_WorkGroupID.y * BLOCK_SIZE;
    float sum = 0.0;
    for (uint k=0; k<TOTAL_K; k+=BLOCK_SIZE) {
        sa[ ly * BLOCK_SZIE + lx ] = src_a.data[ (dy + ly) * src_a_width + ( k+lx) ];
        sb[ ly * BLOCK_SIZE + lx ] = src_b.data[ ( k + ly) * src_b_width + (dx+lx) ];
        barrier();
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum += sa[ ly * BLOCK_SIZE + i ] * sb[ i * BLOCK_SIZE + lx ];
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    dst.data[ (dy+ly) * dst_width + (dx+lx) ] = sum;
}
```

In the barrier, calculate the output element by referring to the shared memory prepared by another thread in the workgroup.

Features of GEMM old implementation

1 thread is responsible for 1 output element

localsize is 8 * 8 = 64

Store A and B together in shared memory

Put a barrier () and refer to the shared memory read by another thread in the workgroup.

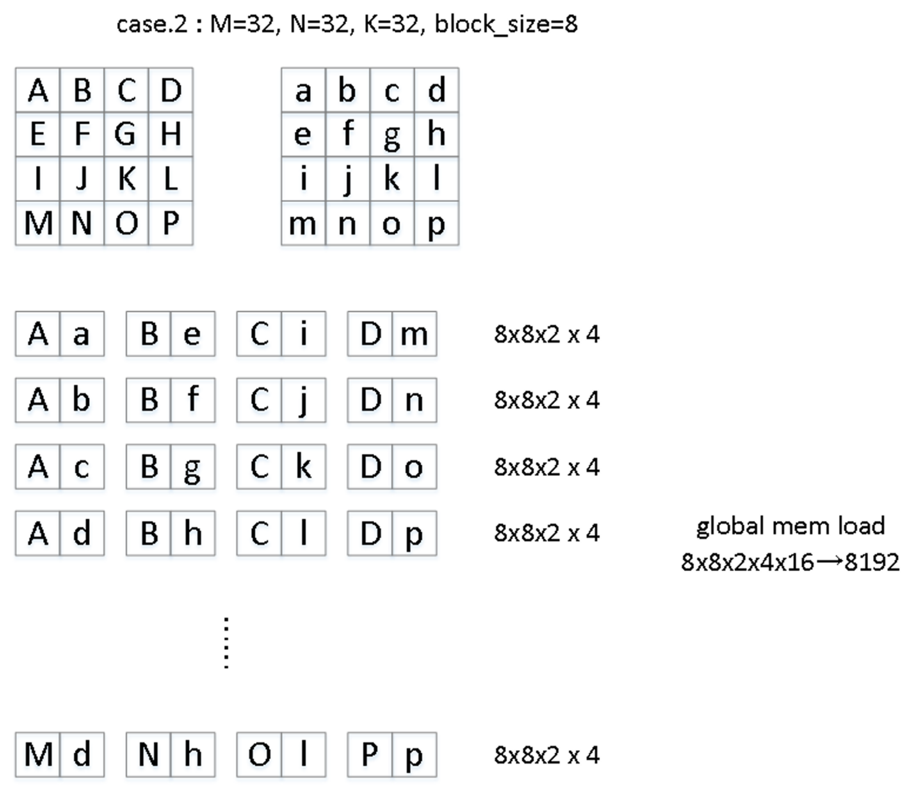
Related research about GEMM

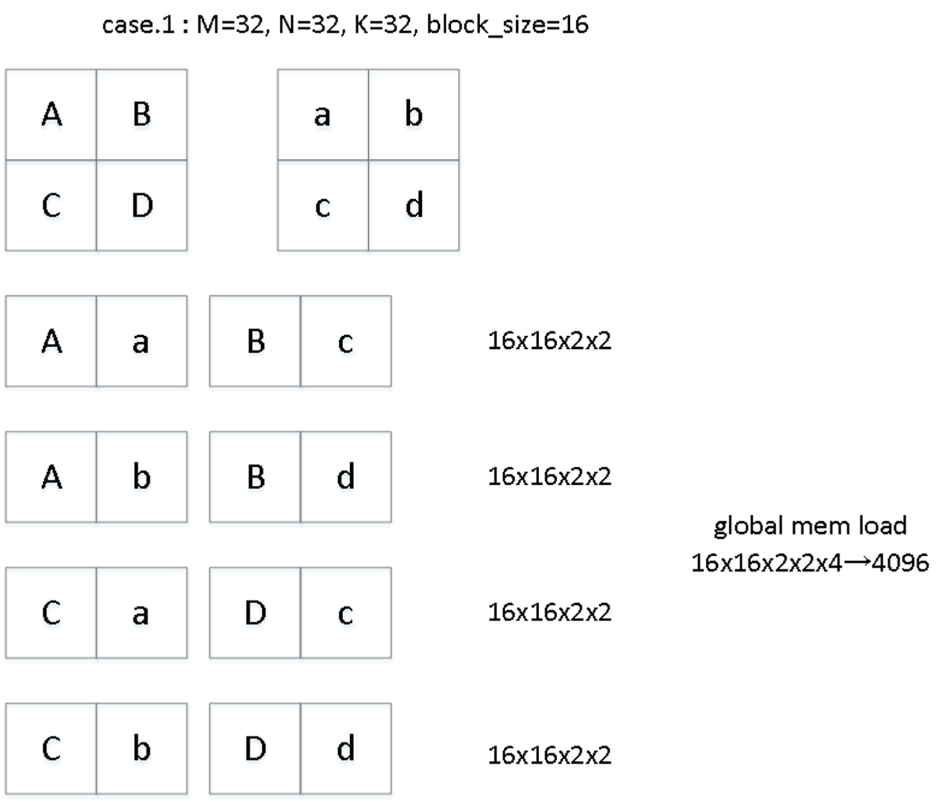
V.Volkov and J.Demmel “Benchmarking GPUs to Tune Dense Linear Algebra”

https://www.cs.colostate.edu/~cs675/volkov08-sc08talk.pdf

- Increasing the localsize will make it easier to cause register spills.
- shared memory is slower than registers
- Putting only B in shared memory and putting A in a register made it faster.

Larger block size reduces global memory access





Optimized implementation of GEMM

```
// I / O definition omitted
#define BLOCK_SIZE 16
layout(local_size_x=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared vec4 sb[ BLOCK_SIZE ];
void main()
{
    // Coordinate calculation part omitted
    float sum[BLOCK_SIZE];
    for (int i=0; i<BLOCK_SIZE; ++i) { sum[i] = 0.0f; }
    for (uint k=0; k<TOTAL_K; k+=4) {
        sb[ lx ] = load_b(dx+lx, k);
        barrier();
        vec4 wa = load_a(dy+lx, k);
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum[i] += dot(wa, sb[i]);
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    for (int i=0; i<BLOCK_SIZE; ++i) {
        dst.data[ (dy+lx) * dst_width + (dx+i) ] = sum[i];
    }
}
```

Optimized implementation of GEMM

```
// I / O definition omitted
#define BLOCK_SIZE 16
layout(local_size_x=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared vec4 sb[ BLOCK_SIZE ];
void main()
{
    // Coordinate calculation part omitted
    float sum[BLOCK_SIZE];
    for (int i=0; i<BLOCK_SIZE; ++i) { sum[i] = 0.0f; }
    for (uint k=0; k<TOTAL_K; k+=4) {
        sb[ lx ] = load_b(dx+lx, k);
        barrier();
        vec4 wa = load_a(dy+lx, k);
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum[i] += dot(wa, sb[i]);
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    for (int i=0; i<BLOCK_SIZE; ++i) {
        dst.data[ (dy+lx) * dst_width + (dx+i) ] = sum[i];
    }
}
```

1 thread is responsible for BLOCK_SIZE output elements

Optimized implementation of GEMM

```
// I / O definition omitted
#define BLOCK_SIZE 16
layout(local_size_x=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared vec4 sb[ BLOCK_SIZE ];
void main()
{
    // Coordinate calculation part omitted
    float sum[BLOCK_SIZE];
    for (int i=0; i<BLOCK_SIZE; ++i) { sum[i] = 0.0f; }
    for (uint k=0; k<TOTAL_K; k+=4) {
        sb[ lx ] = load_b(dx+lx, k);
        barrier();
        vec4 wa = load_a(dy+lx, k);
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum[i] += dot(wa, sb[i]);
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    for (int i=0; i<BLOCK_SIZE; ++i) {
        dst.data[ (dy+lx) * dst_width + (dx+i) ] = sum[i];
    }
}
```

localsize is a one-dimensional thread with BLOCK_SIZE = 16

Optimized implementation of GEMM

```
// I / O definition omitted
#define BLOCK_SIZE 16
layout(local_size_x=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared vec4 sb[ BLOCK_SIZE ];
void main()
{
    // Coordinate calculation part omitted
    float sum[BLOCK_SIZE];
    for (int i=0; i<BLOCK_SIZE; ++i) { sum[i] = 0.0f; }
    for (uint k=0; k<TOTAL_K; k+=4) {
        sb[ lx ] = load_b(dx+lx, k);
        barrier();
        vec4 wa = load_a(dy+lx, k);
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum[i] += dot(wa, sb[i]);
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    for (int i=0; i<BLOCK_SIZE; ++i) {
        dst.data[ (dy+lx) * dst_width + (dx+i) ] = sum[i];
    }
}
```

Only B is stored in shared memory

Optimized implementation of GEMM

```
// I / O definition omitted
#define BLOCK_SIZE 16
layout(local_size_x=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared vec4 sb[ BLOCK_SIZE ];
void main()
{
    // Coordinate calculation part omitted
    float sum[BLOCK_SIZE];
    for (int i=0; i<BLOCK_SIZE; ++i) { sum[i] = 0.0f; }
    for (uint k=0; k<TOTAL_K; k+=4) {
        sb[ lx ] = load_b(dx+lx, k);
        barrier();
        vec4 wa = load_a(dy+lx, k);
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum[i] += dot(wa, sb[i]);
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    for (int i=0; i<BLOCK_SIZE; ++i) {
        dst.data[ (dy+lx) * dst_width + (dx+i) ] = sum[i];
    }
}
```

A is a normal variable and expects register allocation

Optimized implementation of GEMM

```
// I / O definition omitted
#define BLOCK_SIZE 16
layout(local_size_x=BLOCK_SIZE) in;
// shared memory can be commonly referenced from workgroup
shared vec4 sb[ BLOCK_SIZE ];
void main()
{
    // Coordinate calculation part omitted
    float sum[BLOCK_SIZE];
    for (int i=0; i<BLOCK_SIZE; ++i) { sum[i] = 0.0f; }
    for (uint k=0; k<TOTAL_K; k+=4) {
        sb[ lx ] = load_b(dx+lx, k);
        barrier();
        vec4 wa = load_a(dy+lx, k);
        for (uint i=0; i<BLOCK_SIZE; ++i) {
            sum[i] += dot(wa, sb[i]);
        }
        barrier();
    }
    // Exception handling of fractional blocks is omitted
    for (int i=0; i<BLOCK_SIZE; ++i) {
        dst.data[ (dy+lx) * dst_width + (dx+i) ] = sum[i];
    }
}
```

Expected to use 4 arithmetic units from 1 thread with vec4 type packed with 4 floats

Features of new GEMM implementation

1 thread is responsible for the BLOCK_SIZE output element

localsize is BLOCK_SIZE = 16

Only B stored in shared memory

A is a normal variable and expects to use register

Use vec4 from 1 thread to 4 arithmetic units

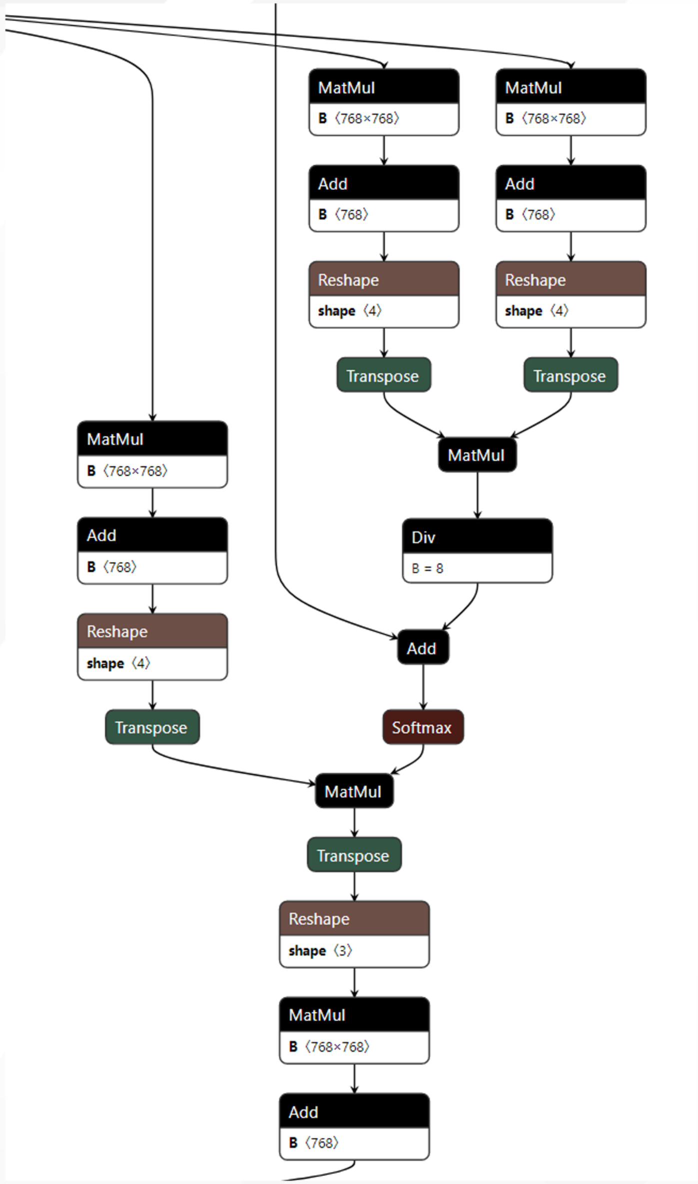
Performance comparison in BERT

BERT is a Transformer-based natural language processing model

The contents are a mass of GEMM = MatMul

Inference time 2.5 to 3.5 times faster with optimized implementation

Based on our benchmark, it is for example 3.53 times faster on Arm Mali G76



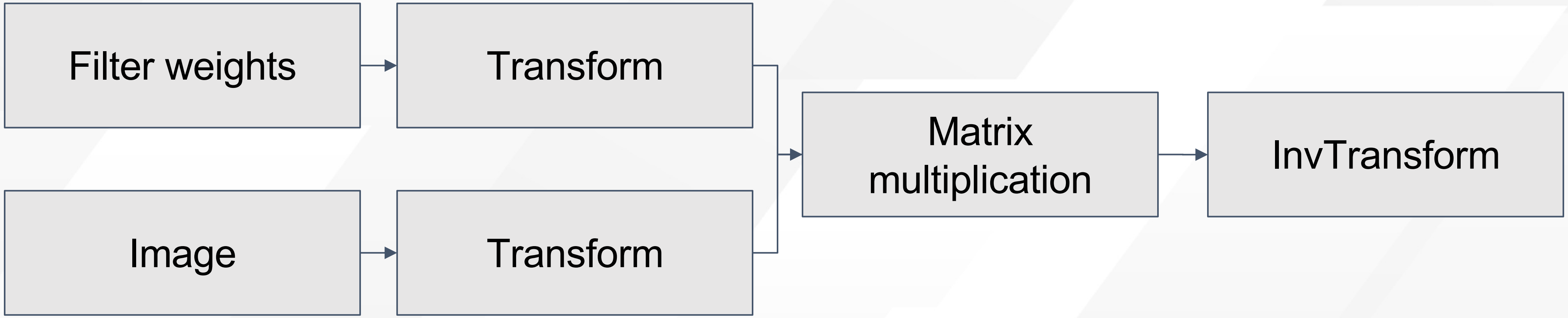
The Winograd Algorithm

The arithmetic complexity of the convolution layer can be reduced by applying transforms to the inputs and the output

Although the transforms are a little demanding, the matrix multiplication which takes up most of the computation time can be reduced

https://arxiv.org/abs/1509.09308

For a 3x3 convolution, the number of multiply operations can be reduced from 36 to 16



Matrix Multiplication Optimization

A GEMM block size is M=N=K=4

(in some architectures M=8)

Computation of the matrix product for each block in every invocation

No synchronization of workgroup

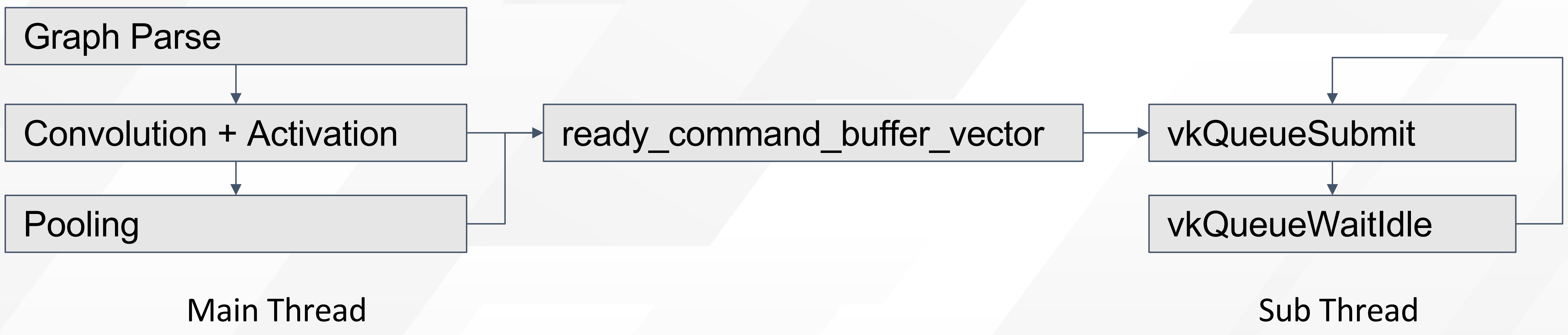
16-bytes memory alignment and reading as vec4 in a single instruction

Memory access optimization by keeping the vector format as much as possible in later processing

Command Buffer Management

The overhead of vkQueueSubmit being large, sub threading is used to reduce the load

1. Start a new thread to process the command buffer
2. When a request is made through ailia's API, the command buffer is added to the sub thread's queue
3. If the sub thread is idle, the command buffer queue is sent all at once to vkQueueSubmit



Summary

Summary

High-performance AI inference can be achieved using NEON and Vulkan regardless of which device or OS and with only standard drivers

Easy integration into any application through ailia SDK

Fast inference using Vulkan has already been implemented for more than 140 AI models

"ailia" resolves all problems

ailia MEDIA

ailia TRAINER

ailia SDK

ailia

ailia MODELS

ailia VISION

ailia APPS

Reference

ax Inc. provides total solutions related to AI, from consulting to model, SDK, AI applications / systems development and support. Should you have any inquiry, feel free to get in touch with us.

ax Inc. home page https://axinc.jp/en/ （Inquiry）

ailia MEDIA https://medium.com/axinc-ai

ailia SDK https://ailia.jp/en/ （Free trial version available）

ailia MODELS https://github.com/axinc-ai/ailia-models

ailia AI Showcase (Video) https://www.youtube.com/watch?v=lRnWX1rDRQU

ailia AI Showcase (Android) https://play.google.com/store/apps/details?id=jp.axinc.ailia_ai_showcase arm

Thank you!

Tweet us: @ArmSoftwareDev -> #AIVTT

Check out our Arm Software Developers YouTube channel

Signup now for our next AI Virtual Tech Talk: developer.arm.com/techtalks

Attendees: don’t forget to fill out the survey to be in with a chance of winning an Arduino Nano 33 BLE board

© 2020 Arm Limited (or its affiliates)


arm DevSummit
Join Arm DevSummit
Live on October 19-21
Where Hardware and Software
Join Forces
Sign Up Today at Devsummit.arm.com

arm AI

AI Virtual Tech Talks Series

Thank You

Danke

Merci

谢谢

ありがとう

Gracias

Kiitos

감사합니다

धन्यवाद

شكرًا

תודה