ncnn - universal neural network inference with vulkan

https://github.com/Tencent/ncnn

nihui, Tencent
About ncnn project

• Since 2017
• Open-source and open collaboration
• High-performance neural network inference framework
• Universal GPU acceleration with Vulkan API
Current status

- Support all major NN frameworks
  - PyTorch, TensorFlow, ONNX, etc.

- Deploy on CPU & GPU across desktop & mobile
  - NVIDIA, AMD, Intel, Apple, ARM, Qualcomm, etc.
  - WebAssembly

- Applications
  - WeChat, QQ, etc.
Some demo

- [https://github.com/nihui/ncnn-android-nanodet](https://github.com/nihui/ncnn-android-nanodet)

- [https://github.com/FeiGeChuanShu/ncnn_nanodet_hand](https://github.com/FeiGeChuanShu/ncnn_nanodet_hand)

- More on [https://github.com/search?q=ncnn](https://github.com/search?q=ncnn)
Vulkan memory management

- One VkBuffer object, multiple offsets
  - Less driver-side Vulkan objects

- Zero-copy on unified memory architecture
  - Common on integrated devices and mobile
Tensor storage layout

- **Wrap the out-most axis:** \([c, h, w] \) to \([c/4, h, w, 4]\)
  - Fast vectorized IO with vec4 storage type

- **Reduce memory bandwidth with fp16 type**
  - VK_KHR_16bit_storage
  - `packHalf2x16()` and `unpackHalf2x16()` as fallback

- **Unified shader source for fp32 / fp16**
  - GLSL macro for storage and arithmetic data type
  - Auto-expand to fp32 / fp16 / etc. based on hardware features

```cpp
afpvec4 v;

v = buf[i]; // fp32 -> fp32 / fp16 -> fp16
v = f16vec4(buf[i]); // fp32 -> fp16
v = vec4(buf[i]); // fp16 -> fp32

// fp16 packed -> fp32
v = vec4(unpackHalf2x16(buf[i].x), unpackHalf2x16(buf[i].y));
// fp16 packed -> fp16
v = f16vec4(unpackHalf2x16(buf[i].x), unpackHalf2x16(buf[i].y));

v = buffer_ld4(buf, i);  
```
Mixed cpu-gpu nn inference

- Optimal data type and layout conversion between GPU and CPU
  - Auto pick the best end-to-end layout conversion pipeline based on hardware features
  - Prefer fp16 data type for transfer on discrete device

<table>
<thead>
<tr>
<th>CPU arch and its optimal layout</th>
<th>upload</th>
<th>GPU and its storage mode</th>
<th>download</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 SSE2 [c/4, h, w, fp32 x 4]</td>
<td></td>
<td>fp32 mode [c/4, h, w, fp32 x 4]</td>
<td></td>
</tr>
<tr>
<td>x86 AVX2 [c/8, h, w, fp32 x 8]</td>
<td></td>
<td>fp16 mode [c/4, h, w, fp16 x 4]</td>
<td></td>
</tr>
<tr>
<td>ARMv7 NEON [c/4, h, w, fp32 x 4]</td>
<td></td>
<td>fp16 x 8 mode [c/8, h, w, fp16 x 8]</td>
<td></td>
</tr>
<tr>
<td>ARMv8.2+fp16 [c/8, h, w, fp16 x 8]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallel inference

- Concurrent vkQueueSubmit()
- Dispatch tasks to multiple devices
  - More GPU utilization and hide VkCommand encoding time
  - Exploit all your computing power :)
Online pipeline cache

● Runtime SPIR-V compilation
  - Binary size increases dramatically in offline way (one glsl source to 10+ SPIR-V variants)
  - We need GLSL source and SPIR-V binary code injection for buggy driver workaround

● Reuse VkPipeline and related vulkan objects
  - NN model has many identical layers usually
  - Hugely reduce the very first loading time (cold boot, without pipeline cache)
Platform specific tricks

- Prefer VkImage on Qualcomm Adreno <= 540
  - Significantly faster with hardware texture fetch
  - Prefer VkBuffer for the rest including Adreno 640+

- Blacklist / whitelist for old-buggy driver
  - Filtered by vendor id + driver version + vulkan api version
  - Report driver bugs as device property info for picking workaround
  - Driver quality improves along with Android system upgrades, especially 8.1 and later :)

![Diagram showing memory management with VkBuffer and VkImage](image)
Swiftshader as the vulkan driver on CPU

- Make sure the vulkan code produces what is expected
- But GLSL source coverage is not possible anyway AFAIK :]

Current building status matrix

<table>
<thead>
<tr>
<th>System</th>
<th>CPU (32bit)</th>
<th>CPU (64bit)</th>
<th>GPU (32bit)</th>
<th>GPU (64bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux (GCC)</td>
<td>build passing</td>
<td>build passing</td>
<td>—</td>
<td>build passing</td>
</tr>
<tr>
<td>Linux (Clang)</td>
<td>build passing</td>
<td>build passing</td>
<td>—</td>
<td>build passing</td>
</tr>
<tr>
<td>Linux (ARM)</td>
<td>build passing</td>
<td>build passing</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Linux (MIPS)</td>
<td>build passing</td>
<td>build passing</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Linux (RISC-V)</td>
<td>—</td>
<td>build passing</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Windows (VS2015)</td>
<td>build passing</td>
<td>build passing</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Windows (VS2017)</td>
<td>build passing</td>
<td>build passing</td>
<td>—</td>
<td>build passing</td>
</tr>
<tr>
<td>Windows (VS2019)</td>
<td>build passing</td>
<td>build passing</td>
<td>—</td>
<td>build passing</td>
</tr>
<tr>
<td>MacOS</td>
<td>—</td>
<td>build passing</td>
<td>—</td>
<td>build passing</td>
</tr>
<tr>
<td>MacOS (ARM)</td>
<td>—</td>
<td>build passing</td>
<td>—</td>
<td>build passing</td>
</tr>
</tbody>
</table>
What we already achieved

ncnn benchmark on Apple M1 Silicon (ms) (less is better)

- CPU (ARMv8.2, fp16, 1 thread)
- GPU (Vulkan, fp16)
Future work

- Vulkan subgroup operation
- VK_NV_cooperative_matrix
Nice to have

- GLSL 8/16-component vector type like vec8, f16vec8, i8vec16, etc.
  - ARM Mali Midgard can do eight 16bit or sixteen 8bit arithmetic

- GLSL builtins for fma(), dot(), int24 arithmetic, etc.
  - Most GPU has the equivalent instructions

- More fine-grained subgroup size info
  - Qualcomm Adreno does fp16 arithmetic with doubled subgroup size
Thanks !